Verification of Second-Order Functional Programs

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

von Diplom-Informatiker Markus Axel Aderhold
aus Darmstadt

zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)

Referenten der Arbeit:	Prof. Dr. Christoph Walther Prof. Dr. Heiko Mantel
Tag der Einreichung:	27.04.2009
Tag der mündlichen Prüfung:	14.07.2009

Darmstadt, 2009

Hochschulkennziffer D17

Abstract

Functional programming languages such as Haskell or ML allow the programmer to implement and to use *higher-order procedures*. A higher-order procedure gets a function as argument and applies this function to some values. For instance, procedure *map* applies a function to all elements of a list and returns the list of the result values.

Verifying that a higher-order program satisfies a certain property is particularly challenging, because it involves reasoning about *indirect* function calls; for instance, a call $map(f, l)$ directly calls procedure *map* and indirectly calls function f if list l is non-empty. Using a higher-order procedure g, a procedure f can be defined by *higher-order recursion*; i. e., procedure f (directly) calls g and passes itself as an argument to g, which leads to indirect recursive calls. These indirect recursive calls make reasoning about higher-order programs difficult.

In this thesis we show how the verification of *second-order* functional programs can be supported by semi-automated theorem provers. *Second-order* means that a procedure such as *map* may only be applied to a first-order function, not to a higher-order function. This suffices for most examples that occur in practice and has advantages concerning the semantics of programs.

Our goal is to verify the *total correctness* of programs. This requires a proof that the program terminates *and* that the result of the computation satisfies a user-defined property. Consequently, we investigate two main problems: termination analysis and inductive theorem proving.

The general contribution of this thesis is the automated analysis of the dynamic call structure in second-order programs (introduced by indirect function calls). Specifically, we describe a technique that automatically analyzes and proves termination of many procedures that occur in practice and a technique that automatically synthesizes induction axioms that are suitable to prove properties of these procedures by induction. Finally, we show how the proof of the base and step cases can be supported by an automated theorem prover.

The techniques have been implemented in the verification tool ✓`eriFun`. Several case studies confirm that they work well in practice and provide a significantly higher degree of automation than other inductive theorem provers.

Zusammenfassung

Funktionale Programmiersprachen wie etwa Haskell oder ML erlauben dem Programmierer, *Prozeduren höherer Ordnung* zu implementieren und zu verwenden. Eine Prozedur höherer Ordnung bekommt eine Funktion als Argument und wendet diese Funktion auf gewisse Werte an. Zum Beispiel wendet die Prozedur *map* eine Funktion auf alle Elemente einer Liste an und gibt die Liste der Ergebniswerte zurück.

Der Nachweis, dass ein Programm höherer Ordnung eine gewisse Eigenschaft erfüllt, birgt besondere Herausforderungen, weil insbesondere auch *indirekte* Funktionsaufrufe zu betrachten sind: Der Aufruf $map(f, l)$ enthält beispielsweise einen direkten Aufruf der Prozedur *map* und indirekte Aufrufe der Funktion f, falls die Liste l nicht leer ist. Unter Verwendung einer Prozedur g höherer Ordnung kann eine Prozedur f durch so genannte *Rekursion höherer Ordnung* definiert werden. Dies bedeutet, dass die Prozedur f (direkt) die Prozedur g aufruft und sich selbst als Argument an g übergibt, was zu indirekten rekursiven Aufrufen führt. Diese indirekten rekursiven Aufrufe erschweren die Beweisführung über Programme höherer Ordnung.

In dieser Dissertation zeigen wir, wie die Verifikation funktionaler Programme *zweiter Ordnung* durch halb-automatische Theorembeweiser unterstützt werden kann. Der Fokus auf Programme *zweiter Ordnung* bedeutet, dass eine Prozedur wie etwa *map* nur auf Funktionen erster Ordnung angewendet werden darf, nicht auf Funktionen höherer Ordnung. Dies genügt für die meisten in der Praxis auftretenden Beispiele und bietet Vorteile hinsichtlich der Semantik von Programmen.

Unser Ziel ist der Nachweis der *totalen Korrektheit* von Programmen. Dies erfordert einen Beweis, dass das Programm terminiert *und* dass das Ergebnis der Berechnung eine vom Benutzer spezifizierte Eigenschaft erfüllt. Dementsprechend untersuchen wir zwei Hauptprobleme: Terminierungsanalyse und Theorembeweisen durch Induktion.

Der allgemeine Beitrag dieser Dissertation ist die automatisierte Analyse der dynamischen Aufrufstruktur von Programmen zweiter Ordnung (verursacht durch indirekte Funktionsaufrufe). Insbesondere beschreiben wir eine Technik, die die Terminierung von zahlreichen in der Praxis auftretenden Prozeduren analysiert und beweist, sowie eine Technik, die automatisch Induktionsaxiome erzeugt, die zum Nachweis von Eigenschaften dieser Proze-

duren mittels Induktion geeignet sind. Schließlich zeigen wir, wie die Beweise der Basis- und Schrittfälle durch einen automatisierten Theorembeweiser unterstützt werden können.

Die vorgestellten Techniken wurden im Verifikationswerkzeug ✓`eriFun` implementiert. Mehrere Fallstudien zeigen, dass sich die Techniken in der Praxis bewähren und einen deutlich höheren Automatisierungsgrad bieten als andere induktive Theorembeweiser.

Wissenschaftlicher Werdegang

1999–2004	Informatikstudium mit Nebenfach Mathematik an der Technischen Universität Darmstadt
09/2002–04/2003	Informatikstudium an der University of British Columbia in Vancouver, Kanada
06/2004	Abschluss des Studiums als Diplom-Informatiker
2004–2009	Promotionsstudium an der Technischen Universität Darmstadt; wissenschaftlicher Mitarbeiter am Fachgebiet Programmiermethodik bei Prof. Dr. Christoph Walther

Contents

List of Figures

List of Tables

Acknowledgments

First of all I would like to thank Christoph Walther for offering me the opportunity to work in his research group. He sparked my interest in automated theorem proving, especially in program verification, and supported my work by sharing with me his deep knowledge of all the details that make automated inductive theorem proving possible in practice. I thank Heiko Mantel for accompanying the final phase of my work as second examiner.

Christoph Benzmüller introduced me to the world of higher-order logic. He gave me valuable feedback on several ideas regarding the verification of higher-order functional programs, which helped me to start off in the right direction. Special thanks go to Alan Bundy for drawing my attention to second-order programs and for asking sharp-sighted questions that helped me to improve the work presented in this thesis.

I wish to thank all participants of the CIAO workshops. These annual meetings have been a great source of inspiration. In particular I thank Lucas Dixon for his lively interest in all aspects of theorem proving, which led to several discussions about what *automated* theorem proving actually means.

It has been pleasant to work with my present and former colleagues at TU Darmstadt. I am grateful for fruitful technical discussions with Stephan Schweitzer, Andreas Schlosser, and Simon Siegler, as well as to Veronika Weber for her support in all sorts of organizational matters. I also thank all developers who contributed to ✓`eriFun`. In particular I am grateful to Nathan Wasser for his work on the implementation concerning second-order procedures and for technical discussions.

Finally, I would like to thank my parents and my sister for their constant support and general encouragement throughout the years.

Chapter 1

Introduction

Today computers are used to solve a large variety of problems: encrypting or decrypting e-mail messages, managing large amounts of data in a database, assisting drivers to keep a car under control in difficult situations such as emergency braking on a slippery road—just to mention a few everyday applications.

Each computer executes *programs*, also known as *software*, that consist of sequences of instructions devised by a programmer. The programmer who writes a program hopes that his program indeed solves the problem the programmer has in mind. In order to gain confidence into his programm, the programmer will *test* it. For instance, a program for encryption and decryption of messages can be tested by encrypting various messages, decrypting them, and checking if each decrypted message coincides with the original message.

Testing is useful to find errors in programs. However, for many problems it is impossible to exhaustively test a program. For example, there are infinitely many different messages that can be encrypted and decrypted. Even if a test has not revealed errors, the program might fail if a message is to be encrypted that, for instance, is written in a foreign language with special characters.

Formal verification is a means to convince oneself (and others) that a program works properly for *all* possible inputs. It requires a formal specification of what a program is supposed to do and then checks if the program satisfies the formal specification. For instance, an encryption/decryption program will offer procedures *encrypt* and *decrypt*. We expect that decryption of an encrypted message yields the original message. Since each message is represented by a sequence of bits, we can assume that a message is a natural number and thus formally specify:

$$\forall msg, key : \mathbb{N}.\ decrypt(encrypt(msg, key), key) = msg \tag{1.1}$$

Unfortunately, the problem of determining if a program satisfies a formal

specification is not semi-decidable. This means that there cannot exist a program *Verify* that, given an object program[1] P and a formal specification ϕ (e. g., $P := \{encrypt, decrypt\}$ and $\phi := (1.1)$), decides within finite time if P satisfies ϕ; moreover, even if P indeed satisfies ϕ, the verifier *Verify* in general cannot find out this fact. This was demonstrated by Kurt Gödel [47] by showing that for any sound formal system there is a true formula ϕ that the formal system is unable to prove.

This negative result carries over to termination analysis of programs: It is impossible to decide if a given object program P will halt for a particular input. This decision problem is called the *halting problem* and has been phrased by Alan Turing [82]. As a consequence, a formal system in general cannot semi-decide *termination*, i. e., if a given object program P will halt for each input. Termination is an important property of many programs. For instance, we expect that a program for encryption and decryption of messages terminates within finite time.

Despite these fundamental limitations it is of course possible to construct formal systems that are able to verify certain properties of programs. The results by Gödel and Turing merely imply that these formal systems are inherently *incomplete*; i. e., they may fail to verify a true property of certain programs. However, they can succeed in many other cases.

Consequently, many tools have been built for formal verification. They differ in the underlying logic and the way proofs are constructed. In particular, they offer varying degrees of automation. These tools are called *theorem provers*, because their goal is to show that a formula follows from some set of axioms. For program verification, the axioms are uniformly extracted from the object program P and capture the semantics of P. The goal is to show that some formula ϕ follows from these axioms. If this is the case, then ϕ is a theorem and it is verified that program P satisfies property ϕ.

In this thesis we consider the verification of second-order functional programs by induction. Our hypothesis is that a degree of automation can be achieved for the verification of *second-order* functional programs that is comparable to the degree of automation that has been achieved for the verification of *first-order* functional programs [18, 33, 36, 37, 56, 57, 58, 93, 95]. Our goal is to verify *total correctness* of second-order programs; i. e., termination of a program *and* compliance with a formally specified property ϕ. In particular, we are concerned with

1. automated termination analysis of second-order programs,
2. automated extraction of "optimized" induction axioms from terminating procedures, and
3. automated proofs of the base and step cases of an inductive proof.

[1] We call the program P that is to be verified the *object program* in order to distinguish it from the verifier *Verify*.

We develop solutions to these problems in the subsequent chapters. These solutions have been integrated into the verification tool ✓eriFun [1, 92, 99] to show that verification of second-order programs can indeed be supported significantly better than it is in current theorem provers.

Section 1.1 motivates second-order programs and informally introduces the functional programming language we use. Then we address the question how properties about second-order programs can be formally specified (Section 1.2) and proved (Section 1.3). In Section 1.4 we briefly review some prominent theorem provers for the verification of programs and show how our contributions improve the state of the art in program verification. Finally, the outline of this thesis is presented in Section 1.5.

1.1 Second-Order Functional Programs

Theoretically, each program can be regarded as a *function* $\varphi : \mathbb{N} \mapsto \mathbb{N}$, because on the hardware level the execution of a program consists of a manipulation of bit sequences, and bit sequences can be interpreted as natural numbers.

Each computable function (i. e., each function that can be computed by a program) can be defined by primitive recursion and μ-recursion [71]. From a practical perspective this "programming language" of primitive recursion and μ-recursion is terribly inconvenient. Therefore high-level programming languages were designed that offer a variety of data types and definition principles for functions. For instance, there are data types for lists, trees, and floating-point numbers. Definition principles for functions include case analyses, loops, and recursion.

Thus *pragmatics* is an important aspect in programming; algorithms should be expressed *naturally* in a programming language. The object language of a theorem prover for the verification of programs needs to be sufficiently rich so that the verification tool is applicable to many verification problems.

However, the object language also needs to be semantically clear. If a programming language does not have a clearly defined semantics, then any attempt of verifying a property of a program written in this language is pointless, because the meaning of the program may change when a different interpreter or compiler is used.

We base our work on a *functional* programming language. The advantage of (purely) functional programming languages is that one does not need to consider side effects (such as the assignment of a value to a variable) or random memory access via pointers.[2] This facilitates the definition of the semantics of programs. In Chapter 2 we formally define syntax and

[2] For the analysis of imperative programs or programs with pointers see [25, 26, 52, 81], for example.

```
structure ℕ <=
  0,
  +(⁻ : ℕ)

structure list[@A] <=
  ε,
  [infixr, 100] ::(hd : @A, tl : list[@A])

procedure len(k : list[@A]) : ℕ <=
if ?ε(k)
  then 0
  else +(len(tl(k)))
end
```

Figure 1.1: A first-order functional program

semantics of the functional programming language we consider. Here we introduce its main features by examples.

Example 1.1. Figure 1.1 shows a first-order functional program. $\mathbb{N}$ denotes the data type of natural numbers. A natural number is either 0 or $^+(n)$ for some natural number $n : \mathbb{N}$. $^+(\ldots)$ denotes the successor function on natural numbers, and 0 and $^+(\ldots)$ are called the *data constructors* of $\mathbb{N}$.

Data structure $list[@A]$ denotes the polymorphic data type of lists. It is polymorphic, because the so-called *type variable* $@A$ can be instantiated with a type so that we can represent lists of natural numbers, lists of trees, lists of lists of natural numbers, etc. Functions ε and :: are the data constructors of *list*, so ε denotes the empty list and $x :: k$ denotes a list that starts with x and continues with another list k. Note that :: is declared as "infix" symbol, so we write $x :: k$ instead of $::(x, k)$. See p. 29 for details.

Each argument position of a data constructor is assigned a *selector* function: $^-(\ldots)$ denotes the predecessor function and is the only selector of data constructor $^+(\ldots)$. Functions hd and tl are the selectors of data constructor :: for lists: $hd(k)$ denotes the first element of a list and $tl(k)$ denotes the sublist of k where the first element has been removed.

Expressions of the form $?cons(t)$ for a data constructor *cons* with selectors $sel_1, \ldots, sel_n$ are equivalent to $t = cons(sel_1(t), \ldots, sel_n(t))$. For example, $?::(k)$ holds iff $k \neq \varepsilon$.

Procedure *len* computes the length of list k. It is defined recursively by computing the length of list $tl(k)$ and then incrementing the result. ♢

The program in Figure 1.1 is a *first-order* program, because procedure *len* operates on *base types*: Both the type $list[@A]$ of parameter k and the return type $\mathbb{N}$ are base types.

```
procedure get.lengths(k : list[list[@NAME]]) : list[ℕ] <=
if ?ε(k)
  then ε
  else len(hd(k)) :: get.lengths(tl(k))
end

procedure sort(k : list[@NAME]) : list[@NAME] <= ...

procedure sort.lists(k : list[list[@NAME]]) : list[list[@NAME]] <=
if ?ε(k)
  then ε
  else sort(hd(k)) :: sort.lists(tl(k))
end
```

Figure 1.2: Some procedures on lists

In addition to base types, a second-order program can also contain procedures with parameters of *function types*. The following example motivates the use of second-order programs.

Example 1.2. Imagine a tourist office that offers guided tours. A particular guided tour takes place several times a day. For each time the tourist office keeps a list of the names of the registered participants. All in all, the tourist office maintains a list of lists of names, one list for each time a tour takes place.

In order to make sure that the groups are neither too big nor too small, the tour guide would like to know how many people registered for each tour. Procedure *get.lengths* of Figure 1.2 accomplishes this task and returns a list of group sizes by applying procedure *len* to each list of names.

Since it is faster to check off the participants that show up for the tour if each list of names is sorted alphabetically, Figure 1.2 also provides a procedure *sort.lists* to apply some sorting procedure *sort* to each list of names. ♢

Procedures *get.lengths* and *sort.lists* are structurally identical, because both procedures iterate over a list, apply some procedure to each element, and return the list of the results. Duplicating code is bad programming style, so good programmers avoid writing procedures as in Figure 1.2 if the programming language offers alternatives.

A widely used approach to avoid redundancy as in procedures *get.lengths* and *sort.lists* is the use of *second-order procedures*. A second-order procedure is a procedure that gets a first-order function as argument. Figure 1.3 lists several common second-order procedures. In general, a function is of order n if its input and output types are at most of order $n-1$. Base types

```
procedure map(f : @A → @B, k : list[@A]) : list[@B] <=
if ?ε(k)
  then ε
  else f(hd(k)) :: map(f, tl(k))
end

procedure every(p : @A → bool, k : list[@A]) : bool <=
if ?ε(k)
  then true
  else if p(hd(k))
         then every(p, tl(k))
         else false
       end
end

procedure some(p : @A → bool, k : list[@A]) : bool <=
if ?ε(k)
  then false
  else if p(hd(k))
         then true
         else some(p, tl(k))
       end
end

procedure filter(p : @A → bool, k : list[@A]) : list[@A] <=
if ?ε(k)
  then ε
  else if p(hd(k))
         then hd(k) :: filter(p, tl(k))
         else filter(p, tl(k))
       end
end
```

Figure 1.3: Frequently used second-order procedures

such as $\mathbb{N}$, $list[@A]$, and $list[\mathbb{N}]$ are of order 0. Type *bool* is a predefined base type of truth values *true* and *false*. We give a formal definition of the order of types and terms in Chapter 2. (Note that we use the term *procedure* for functions that are defined by an implementation. Thus *len* is a procedure, whereas :: is a function.)

For instance, procedure *map* applies a first-order function f to each element of list k and returns the list of the results:

$$map(f,\ a :: b :: c :: \varepsilon) = f(a) :: f(b) :: f(c) :: \varepsilon$$

```
procedure foldl(f : @A × @B → @A, x : @A, k : list[@B]) : @A <=
if ?ε(k)
  then x
  else foldl(f, f(x, hd(k)), tl(k))
end

procedure foldr(f : @A × @B → @B, x : @B, k : list[@A]) : @B <=
if ?ε(k)
  then x
  else f(hd(k), foldr(f, x, tl(k)))
end

procedure rev_itlist(f : @A × @B → @B, x : @B, k : list[@A]) : @B <=
if ?ε(k)
  then x
  else rev_itlist(f, f(hd(k), x), tl(k))
end
```

Figure 1.4: Second-order *fold* procedures

This is exactly the common pattern of procedures *get.lengths* and *sort.lists*. Thus instead of implementing the procedures of Figure 1.2, it is better to implement procedure *map* and to write

$$map(len, k) \quad \text{instead of} \quad get.lengths(k) \quad \text{and}$$
$$map(sort, k) \quad \text{instead of} \quad sort.lists(k) \ .$$

Similarly to *map*, the other second-order procedures in Figure 1.3 also iterate over a list and compute the following:

- $every(p,\ a :: b :: c :: \varepsilon) \leftrightarrow p(a) \wedge p(b) \wedge p(c)$
- $some(p,\ a :: b :: c :: \varepsilon) \leftrightarrow p(a) \vee p(b) \vee p(c)$
- *filter*(p, k) returns a copy of list k where all elements x have been removed that do not satisfy $p(x)$.

Figure 1.4 lists *fold* procedures that are frequently used in functional programming.[3] The semantics of these procedures (both in prefix and infix notation) is summarized in Table 1.1. For example, if $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ denotes addition of natural numbers, then $foldl(+, 0, k)$ computes the sum of all elements of list $k : list[\mathbb{N}]$.

[3]Procedure *foldr* is sometimes called *itlist*: "iteratively apply a binary function between adjacent elements of a list".

Table 1.1: Semantics of the *fold* procedures in Figure 1.4

procedure call	result (prefix not.)	result (infix not.)
$foldl(\circ,\ x, a :: b :: c :: \varepsilon)$	$\circ(\circ(\circ(x, a), b), c)$	$((x \circ a) \circ b) \circ c$
$foldr(\circ,\ x, a :: b :: c :: \varepsilon)$	$\circ(a, \circ(b, \circ(c, x)))$	$a \circ (b \circ (c \circ x))$
$rev\_itlist(\circ,\ x, a :: b :: c :: \varepsilon)$	$\circ(c, \circ(b, \circ(a, x)))$	$c \circ (b \circ (a \circ x))$

The next examples demonstrate the use of two further salient features of second-order programs: second-order recursion and λ-expressions.

Example 1.3. Figure 1.5 defines a data structure $term[@V, @F]$ that represents terms.[4] A term t is either a variable or the application of a function symbol to a list of arguments, which themselves are terms.

Terms that do not contain variables are commonly called *ground terms*. Procedure *groundterm* checks if a term t is a ground term: A variable is not a ground term, so *groundterm* returns *false* in this case. If t is of the form $t = apply(fsym(t), args(t))$, then *groundterm* checks if each argument term is a ground term by passing itself as argument to *every*. ♢

Example 1.4. Procedure *subterm* checks if a term r occurs as subterm of term t: If $r = t$, then r clearly occurs in t. Otherwise, r cannot be a subterm of t if t is a variable. If t is of the form $t = apply(fsym(t), args(t))$, then procedure *subterm* checks if r occurs in some argument term. To this end we use the λ-expression $\lambda s : term[@V, @F].\ subterm(r, s)$ that denotes a function $p : term[@V, @F] \to bool$ with $p(s) \leftrightarrow subterm(r, s)$. ♢

Programs with second-order recursion are particularly difficult to analyze, because one additionally needs to reason about *indirect* recursive calls. For instance, procedure *groundterm* contains no *direct* recursive call. It calls itself indirectly via a call of *every*. The goal of this thesis is to facilitate the automated analysis of second-order programs that may use second-order recursion.

In programming languages such as Haskell [54] and ML [70] the second-order procedures of Figures 1.3 and 1.4 are considered as *higher-order* procedures. For example, procedure *map* could also apply a second-order function to a list of first-order functions. We give examples of possible applications of higher-order procedures in Section 2.2 and also discuss the arising problems.

[4] Data structure $term[@V, @F]$ does not ensure that terms are *well-formed*, i. e., that the number n of arguments of each function call $f(t_1, \ldots, t_n)$ in a term t corresponds to the arity of $f : @F$ (given by some signature). Such a check can be performed by a procedure *wellformed* as in Appendix A.2.

```
structure term[@V, @F] <=
  var(vsym : @V),
  apply(fsym : @F, args : list[term[@V, @F]])

procedure groundterm(t : term[@V, @F]) : bool <=
case t of
  var   : false,
  apply : every(groundterm, args(t))
end

procedure subterm(r, t : term[@V, @F]) : bool <=
if r = t
  then true
  else case t of
         var   : false,
         apply : some(λs : term[@V, @F]. subterm(r, s), args(t))
       end
end
```

Figure 1.5: Second-order recursion in procedures *groundterm* and *subterm*

1.2 Specifying Properties of Programs

We specify properties of programs by *formulas*. For instance, we expect *map* to return a list of the same length as the input list, regardless of the function f that is applied to the elements of list k:

$$\forall f : @A \to @B,\ k : list[@A].\ len(map(f, k)) = len(k) \tag{1.2}$$

This is a formula of *third-order logic*: It quantifies over variables of order ≤ 1 (because f is of order 1 and k is of order 0) and uses function symbols of order ≤ 2 (because *map* is of order 2 and *len* is of order 1).

Table 1.2 shows the general definition of the order of a logic [14, 15, 60]. Thus in order to specify a property of a program of order m, one should consider a logic of order $2m - 1$. In a formula of such a logic, all functions of order m may occur and quantification is allowed over all inputs of these functions.

Example 1.5. Procedure *subterm* should compute a transitive relation. We can formally specify this by:

$$\forall t_1, t_2, t_3 : term[@V, @F].$$
$$subterm(t_1, t_2) \wedge subterm(t_2, t_3) \to subterm(t_1, t_3)$$

Table 1.2: The order of a logic

order of the logic	variables and constants of order	quantified variables of order
0	0	0
1	≤ 1	≤ 0
2	≤ 1	≤ 1
3	≤ 2	≤ 1
4	≤ 2	≤ 2
$2m-1$	$\leq m$	$\leq m-1$
$2m$	$\leq m$	$\leq m$

The following formula states that a ground term does not contain a variable as subterm:

$$\forall v, t : term[@V, @F].\ ?var(v) \wedge groundterm(t) \rightarrow \neg\, subterm(v, t) \qquad \diamondsuit$$

We restrict the language of formulas to *universally quantified* formulas, so all formulas are of the form $\forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ b$ for variables $x_1, \ldots, x_n$, types $\tau_1, \ldots, \tau_n$, and a Boolean term b. Instead of specifying a property

$$\forall x.\ \exists y.\ P(x, y)\,, \tag{1.3}$$

one therefore needs to specify this property constructively by supplying a so-called *Skolem function* f:

$$\forall x.\ P(x, f(x)). \tag{1.4}$$

A proof of (1.4) is *constructive* in the sense that for every x it clearly exhibits some y that satisfies $P(x, y)$, namely $y := f(x)$.

The restriction to universally quantified formulas simplifies the calculus for the proof of formulas. Additionally one could allow existential quantification and add special support to handle existential quantifiers. This special support may comprise constructive proof rules that enable the user to prove existential formulas without explicitly specifying an implementation of the Skolem function. This may as well lead to a constructive proof from which one can automatically extract an implementation of the Skolem function, see [27] for example. Since a constructive proof of (1.3) requires the synthesis of a Skolem function f (either explicitly or implicitly via extraction from a proof), a theorem prover could also support the synthesis of f by using *algorithm libraries* that contain patterns for certain algorithms (e. g., the principle of divide-and-conquer) [35].

Since our focus is the verification of *existing* programs, we do not consider such synthesis problems in this thesis and stick to universally quantified formulas.

```
procedure [infixr, 20] +(x, y : ℕ) : ℕ <=
if ?0(x)
  then y
  else +(⁻(x) + y)
end

procedure dbl(n : ℕ) : ℕ <=
if ?0(n)
  then 0
  else +(+(dbl(⁻(n))))
end
```

Figure 1.6: Procedures + and *dbl* for addition and multiplication by 2

1.3 Verification of Second-Order Programs

Induction is a widely used technique to prove universally quantified formulas. It can be used whenever there is a well-founded relation on the domain that the quantification ranges over.

Example 1.6. We can prove a formula $\forall n : \mathbb{N}.\ goal[n]$ by induction on n by showing

- the base case $goal[\mathtt{0}]$ and
- the step case $\forall n : \mathbb{N}.\ n \neq \mathtt{0} \wedge goal[^{-}(n)] \rightarrow goal[n]$.

In other words, we prove $\forall n : \mathbb{N}.\ goal[n]$ using the following *induction axiom*:

> For any formula P that contains a free variable $n : \mathbb{N}$:
> $P[\mathtt{0}] \wedge \big(\forall n : \mathbb{N}.\ n \neq \mathtt{0} \wedge P[^{-}(n)] \rightarrow P[n]\big) \rightarrow \forall n : \mathbb{N}.\ P[n]$.

Premises $P[\mathtt{0}]$ and $\forall n : \mathbb{N}.\ n \neq \mathtt{0} \wedge P[^{-}(n)] \rightarrow P[n]$ of the induction axiom are typically called the *induction formulas* for $\forall n : \mathbb{N}.\ P[n]$. Premise $P[^{-}(n)]$ in the second induction formula is called the *induction hypothesis*. ♢

The induction axiom of Example 1.6 is well suited to prove a formula such as

$$\forall n : \mathbb{N}.\ dbl(n) = n + n$$

about procedures + and *dbl* in Figure 1.6, because both procedures contain direct recursive calls with arguments $^{-}(x)$ and $^{-}(n)$, respectively. These direct recursive calls correspond to the induction hypothesis $P[^{-}(n)]$ of the induction axiom.

Theorem provers for proofs by induction typically extract the induction axiom of Example 1.6 from the definition of procedures + and *dbl*: First,

they prove that all procedures $f : \tau_1 \times \ldots \times \tau_n \to \tau$ terminate. Hence the relation $\succ_f$ on $\tau_1 \times \ldots \times \tau_n$, defined by $(x_1, \ldots, x_n) \succ_f (x'_1, \ldots, x'_n)$ iff evaluation of $f(x_1, \ldots, x_n)$ requires the evaluation of a recursive call $f(x'_1, \ldots, x'_n)$, is well-founded; i. e., there is no infinite sequence

$$(x_1, \ldots, x_n) \succ_f (x'_1, \ldots, x'_n) \succ_f (x''_1, \ldots, x''_n) \succ_f \ldots$$

Thus it is sound to prove a formula by well-founded induction wrt. $\succ_f$. Example 1.6 is the induction axiom for well-founded induction wrt. $\succ_{dbl}$.

Example 1.7. We can prove a formula $\forall t : \mathit{term}[@\,V, @F].\ \mathit{goal}[t]$ by induction on t by showing

- the base case $\forall t : \mathit{term}[@\,V, @F].\ ?\mathit{var}(t) \to \mathit{goal}[t]$ and
- the step case

$$\begin{aligned}&\forall t : \mathit{term}[@\,V, @F].\\ &?\mathit{apply}(t) \wedge \big(\forall s : \mathit{term}[@\,V, @F].\ s \in \mathit{args}(t) \to \mathit{goal}[s]\big) \to \mathit{goal}[t]\,.\end{aligned}$$

The base case comprises all variables $t = \mathit{var}(\mathit{vsym}(t))$, whereas the step case comprises all terms $t = \mathit{apply}(\mathit{fsym}(t), \mathit{args}(t))$. In the step case we may use the induction hypothesis that $\mathit{goal}[s]$ holds for all direct subterms s of t. $\diamondsuit$

Note that the induction hypothesis

$$\forall s : \mathit{term}[@\,V, @F].\ s \in \mathit{args}(t) \to \mathit{goal}[s] \tag{1.5}$$

in Example 1.7 is fairly complex from a computational point of view: It quantifies over *infinitely* many terms $s : \mathit{term}[@\,V, @F]$ and afterwards constrains s to be an element of list $\mathit{args}(t)$. Alternatively, we can phrase the induction hypothesis as

$$\mathit{every}(\lambda s : \mathit{term}[@\,V, @F].\ \mathit{goal}[s],\ \mathit{args}(t))\,. \tag{1.6}$$

This is the form we aim for in this thesis, because it exhibits the quantification over a *finite* domain more clearly than (1.5).

The induction axiom of Example 1.7 is useful to prove a formula such as

$$\forall v, t : \mathit{term}[@\,V, @F].\ ?\mathit{var}(v) \wedge \mathit{groundterm}(t) \to \neg\, \mathit{subterm}(v, t)\,.$$

To a human it might seem suggestive that the induction axiom of Example 1.7 is suitable for a proof about procedures *groundterm* and *subterm*. However, it is not obvious how this induction axiom can be extracted from the definition of procedures *groundterm* and *subterm*. We show a solution to this problem in Chapter 5.

1.4 A Brief Overview of some Theorem Provers

The following list of theorem provers is not meant to be a complete overview of all theorem provers that have been developed so far. However, it contains the most prominent theorem provers:

ACL2 ACL2's first-order, quantifier-free[5] logic of recursive functions is based on the applicative subset of Common Lisp and proves theorems by well-founded induction [2, 59].

Coq Coq is an interactive proof assistant for the development of mathematical theories and formally certified software. It is based on a theory called the *calculus of inductive constructions*, a variant of type theory [3, 29].

HOL HOL is an interactive proof assistant for higher-order logic. It offers some built-in decision procedures and subsidiary theorem provers that can automatically establish some theorems [4, 49].

Isabelle Isabelle is a generic system for implementing logical formalisms. Isabelle/HOL is the specialization of Isabelle to higher-order logic [5, 66].

Nuprl The logic of Nuprl is a constructive type theory similar to Martin-Löf type theory. Nuprl is a proof assistant that provides a metalanguage for "proof-generating programs"; i. e., the user writes a metalanguage script that generates a proof which is checked by Nuprl [6, 40].

PVS PVS is based on classical, typed higher-order logic. Its inference mechanisms are applied interactively under user guidance [7, 67].

✓eriFun Similarly to ACL2, ✓eriFun is based on first-order, quantifier-free logic. The specification language is a functional programming language and theorems are proved by well-founded induction [1, 92].

The users of theorem provers can roughly be divided into two groups:

- The first group comprises experts who use a theorem prover just to manage the complexity of a proof. For instance, there may be a lot of cases to consider and some cases may involve tricky calculations. The theorem prover makes sure that no case is forgotten and that there are no mistakes in the reasoning steps.

  These users want to keep full control over the proof. Thus they are prepared to provide the theorem prover with a detailed proof script.

[5] *Quantifier-free* means that formulas are not *explicitly* universally quantified as in Section 1.2, but *implicitly* universally quantified. In other words, formulas are written without the leading universal quantifier.

The theorem prover checks this proof script and just fills in the (small) blanks that would be a cumbersome routine job for the user.

- The second group comprises users who expect the theorem prover to prove as much as possible automatically. The design of such theorem provers is particularly challenging: Not only do the theorem provers need to "fill in the details", but they additionally need to generate a proof plan from scratch.

  Theorem provers for this group of users need to make many more decisions than theorem provers that mainly check detailed proof scripts. Due to their high degree of automation, however, they are also appealing to less experienced users.

Our development focuses on the second group of users; i. e., users who expect a high degree of automation from a theorem prover. Experience from teaching [95] shows that highly automated theorem provers are especially attractive to students (and students are the programmers of today and tomorrow): Often students find it difficult to figure out how to start off with a proof. Hence if a theorem prover heuristically suggests some proof steps, it is much easier for them to consider and pursue these initial attempts than having to prove everything on their own.

Highly automated theorem provers offer a combination of the following features to support the development of proofs (this list extends the list of features from p. 2):

1. automated termination analysis (so that the user can quickly move on to verify a property ϕ of a program)

2. automated extraction of induction axioms and heuristic selection of a suitable induction axiom for a proof of a given formula (so that the user easily gets started with the proof)

3. automated proofs of the base and step cases (so that the user does not need to prove all the details by himself)

4. a heuristic that analyzes unfinished proof attempts and suggests auxiliary lemmas that help to complete the proof (so that proof attempts of base and step cases are completed)

5. a disprover that searches for counterexamples for a given formula (so that the user does not waste time with trying to prove a false conjecture)

ACL2 supports all features except the disprover [33, 42]. ✓`eriFun` provides support for all five features [10, 11, 13, 73, 80, 85, 86]. Both tools

directly address the verification of programs written in a functional programming language, but are limited to *first-order* programs. Consequently they cannot be used to verify second-order programs.

Automation of inductive proofs along the lines of features 1–4 has been integrated into Isabelle by the IsaPlanner [41]; a disprover for Isabelle (i. e., feature 5) is described in [28]. The problem with Isabelle wrt. program verification is that higher-order logic is not a programming language. While higher-order logic is of course expressive enough to state the defining equations of a second-order program, it does not provide an operational semantics for these defining equations. This lack of an operational semantics has severe consequences: The transformation of HOL equations into a program [51] yields programs with different semantics depending on the chosen target language. Even if "termination" of a "HOL program" has been proved in Isabelle, the resulting ML or Haskell program need not terminate [62]. A further drawback of Isabelle is that termination analysis of programs with second-order recursion requires user interaction and easily leads to suboptimal induction axioms, see Chapter 6. Isabelle's focus is on the formalization of mathematics, not the verification of programs.

Our contributions to program verification focus on a functional programming language with a *call-by-value* semantics and have been integrated into ✓`eriFun`.

1.5 Thesis Outline

We define syntax and semantics of the functional programming language $\mathcal{L}$ we consider in Chapter 2. This includes a discussion of the expressive power of the programming language and the introduction of some terminology that we use in the subsequent chapters. In particular, we discuss the meaning of *termination*.

Chapter 3 investigates finite (universal) quantification. Section 1.3 already showed that finite quantification occurs in induction hypotheses of induction axioms that are derived from procedures with second-order recursion. We look at two kinds of *quantification procedures* (these are decision procedures for finite quantifications): quantification procedures for data structures (such as quantification over all elements of a list) and quantification procedures for second-order procedures (which quantify over the function calls a second-order procedure initiates).

Chapter 4 is devoted to the first aspect of total correctness: termination. We develop techniques for *interactive* termination proofs of procedures (in particular ones with second-order recursion) and for *automated* termination proofs (again with special emphasis on second-order recursion). Our automated approach extends the method of *argument-bounded functions* [86, 96] in two respects: Firstly, it also inspects *components of types*. Secondly, it

adds a facility to take care of second-order recursion. The extension maintains an important advantage of the original approach; it yields information that helps to "optimize" (i. e., generalize) induction axioms.

Chapter 5 revolves around proving properties of second-order programs by induction, i. e., the second aspect of total correctness. First, we describe the synthesis and representation of well-founded relations from data structure definitions and terminating procedures. We show how the information obtained from termination analysis can be used to enlarge the relations of terminating procedures, which generalizes the corresponding induction axioms. In order to prove base and step cases of inductive proofs about second-order programs, we extend ✓`eriFun`'s calculus for *symbolic evaluation* (i. e., evaluation of terms that contain variables). An example proof demonstrates how the different components of our approach work together.

Chapter 6 compares our approach with related work. We evaluate our approach in Chapter 7. Here we also discuss some alternatives that may be useful for porting our approach to other theorem provers. We conclude with an outlook on further research directions in Chapter 8. The appendix lists some verified example programs that illustrate applications of our approach.

Finally, we provide two indices: The first index comprises the function symbols and type symbols we use in examples (e. g., *groundterm* and $list[@A]$). The second index is the general index of terms and symbols we use in this thesis. It begins with all symbols and symbol-like expressions (e. g., $\rhd$ and $\mathcal{CL}(\Sigma, \mathcal{V})$). Afterwards, it lists the "natural language terms" in alphabetical order.

Chapter 2

The Object Language $\mathcal{L}$

In the following sections, we define syntax and semantics of the functional programming language $\mathcal{L}$ 2.0 that the programs to be verified are written in. $\mathcal{L}$ 2.0 extends the functional programming language $\mathcal{L}$ 1.0 that is used in ✓`eriFun` 3.2.2 [90] from a first-order programming language to a second-order programming language.

Like $\mathcal{L}$ 1.0, $\mathcal{L}$ 2.0 features an ML-style polymorphic type system [38, 73, 90]. For instance, one can define a type $\mathit{list}[@A]$ for polymorphic lists (also called *generic lists*). Here $@A$ is a type variable that may be instantiated with a type such as $\mathbb{N}$ to represent lists of natural numbers, $\mathit{list}[\mathbb{N}]$.

$\mathcal{L}$ 2.0 extends $\mathcal{L}$ 1.0 by supporting

- function types $\tau_1 \times \ldots \times \tau_k \to \tau_{k+1}$ and
- λ-expressions $\lambda x_1 : \tau_1, \ldots, x_n : \tau_n . t$.

Differently to ML, a type variable $@A$ may not be instantiated with a function type. We discuss this restriction in Section 2.2.

In the following, we refer to $\mathcal{L}$ 2.0 simply by $\mathcal{L}$.

Organization of this chapter. In Section 2.1 we define the syntax of types, terms, data structure definitions, procedure definitions, and lemma definitions in $\mathcal{L}$ as well as some general terminology that we use in the subsequent chapters. We discuss the expressive power of $\mathcal{L}$ in Section 2.2. The semantics of $\mathcal{L}$-programs is defined in Section 2.3. Finally, Section 2.4 introduces a simplifying notation for procedures.

2.1 Syntax

Each $\mathcal{L}$-program P defines a type signature $\Omega(P)$ and a term signature $\Sigma(P)$. The idea is to start with an empty program P that defines initial signatures $\Omega(P) := \Omega_{init}$ and $\Sigma(P) := \Sigma_{init}$ (cf. Definition 2.10 on p. 21). Then we

add a definition of a data structure (Definition 2.31 on p. 30), of a procedure (Definition 2.39 on p. 36), or of a lemma (Definition 2.46 on p. 39). Each definition extends program P to a program $P' \supset P$ and correspondingly extends the type signature and term signature to $\Omega(P')$ and $\Sigma(P')$, respectively.

We begin with some general notation for types and terms, respectively. Most of these definitions are standard and can be found in introductory textbooks, see [15, 20, 88] for example. Sometimes our definitions are a bit more restrictive due to our focus on second-order programs.

2.1.1 Types

We assume a countably infinite set $\mathcal{W}$ of *type variables*. All type variables start with prefix @; e. g., $@A_1, @A_2, \ldots, @B, @C, \ldots$ A *type signature* $\Omega = (\Omega_k)_{k\in\mathbb{N}}$ is a family of pairwise disjoint sets of symbols that denote *type constructors*. For a type constructor $str \in \Omega_k$, we say that *str* has *arity* k.

Each type signature Ω is assumed to contain a type constructor $bool \in \Omega_0$ to denote the type of truth values (*true* and *false*).[1] Type *bool* is special (because this type is used for *predicates*, see p. 23), so a type variable $@A$ may *not* be instantiated with *bool*. This does not reduce the expressivity of $\mathcal{L}$, because the programmer is free to define another type *mybool* that type variables may be instantiated with.

Definition 2.1 (Types)**.** *The family* $Types(\Omega, \mathcal{W}) = (Types(\Omega, \mathcal{W})_n)_{n\in\mathbb{N}}$ *of* types *is the smallest family of sets that satisfies the following requirements:*

- $\mathcal{W} \subseteq Types(\Omega, \mathcal{W})_0$*, i. e., each type variable is a type.*
- *If* $\tau_1, \ldots, \tau_k \in Types(\Omega, \mathcal{W})_0 \setminus \{bool\}$ *and* $str \in \Omega_k$*, then*

  $$str[\tau_1, \ldots, \tau_k] \in Types(\Omega, \mathcal{W})_0 \,.$$

- *If* $\tau_1, \ldots, \tau_k, \tau_{k+1}$ *are types,* $k \geq 1$*, such that* $\tau_i \in Types(\Omega, \mathcal{W})_{n_i}$ *for all* $i = 1, \ldots, k+1$*, some* $n_i \in \mathbb{N}$*, and* $\tau_i \neq bool$ *for all* $i = 1, \ldots, k$*, then*

  $$\tau_1 \times \ldots \times \tau_k \to \tau_{k+1} \in Types(\Omega, \mathcal{W})_m \,,$$

  where $m := 1 + \max\{n_1, \ldots, n_{k+1}\}$.

The order *of a type* $\tau \in Types(\Omega, \mathcal{W})_n$ *is defined as* n*. Types of order 0 are called* base types*. Types of order* $n \geq 1$ *are called* function types*. A* ground type *is a type that does not contain type variables. We use* $Types(\Omega)$ *to*

[1] In type theory (see [15], for example), the type of truth values is often denoted by o. We write *bool* instead of o to follow the convention of theorem provers like Isabelle [66], PVS [68], and ✓eriFun [90].

denote the family of ground types. If the order n of a type $\tau \in \mathit{Types}(\Omega, \mathcal{W})_n$ does not matter, we simply write $\tau \in \mathit{Types}(\Omega, \mathcal{W})$ or $\tau \in \mathit{Types}(\Omega)$. A base type $\tau \in \mathit{Types}(\Omega, \mathcal{W})_0$ is called polymorphic *iff τ contains at least one type variable. Otherwise τ is called* monomorphic.

Example 2.2. If $\mathbb{N} \in \Omega_0$, $\mathit{list} \in \Omega_1$, and $\mathit{pair} \in \Omega_2$, then:

- $@A \in \mathit{Types}(\Omega, \mathcal{W})_0$ and $\mathbb{N} \in \mathit{Types}(\Omega, \mathcal{W})_0$
- $\mathit{list}[@A] \in \mathit{Types}(\Omega, \mathcal{W})_0$ and $\mathit{list}[\mathbb{N}] \in \mathit{Types}(\Omega, \mathcal{W})_0$
- $\mathit{pair}[@A, @B] \in \mathit{Types}(\Omega, \mathcal{W})_0$ and $\mathit{pair}[\mathbb{N}, \mathit{list}[\mathbb{N}]] \in \mathit{Types}(\Omega, \mathcal{W})_0$
- $\mathit{list}[@A] \to @A \in \mathit{Types}(\Omega, \mathcal{W})_1$
- $\mathbb{N} \times \mathit{list}[\mathbb{N}] \to \mathbb{N} \in \mathit{Types}(\Omega, \mathcal{W})_1$
- $\mathbb{N} \times (\mathit{list}[\mathbb{N}] \to \mathbb{N}) \to \mathbb{N} \in \mathit{Types}(\Omega, \mathcal{W})_2$
- $\mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \in \mathit{Types}(\Omega, \mathcal{W})_2$
- $(\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \in \mathit{Types}(\Omega, \mathcal{W})_2$
- $((\mathit{list}[\mathbb{N}] \to \mathbb{N}) \to \mathbb{N}) \to \mathbb{N} \in \mathit{Types}(\Omega, \mathcal{W})_3$ ♢

If a type τ is composed of other types, we call these other types the *components* of type τ. These components occur at a certain position in τ:

Definition 2.3 (Type positions)**.** *For a type $\tau \in \mathit{Types}(\Omega, \mathcal{W})$, $\mathit{Pos}(\tau) \subset \mathbb{N}^*$ denotes the set of all* type positions of τ.[2] *It is defined as the smallest set that satisfies the following requirements:*

- $\epsilon \in \mathit{Pos}(\tau)$
- $h\pi \in \mathit{Pos}(\tau)$ *if* $\tau = \mathit{str}[\tau_1, \ldots, \tau_k]$, $h \in \{1, \ldots, k\}$, *and* $\pi \in \mathit{Pos}(\tau_h)$
- $h\pi \in \mathit{Pos}(\tau)$ *if* $\tau = \tau_1 \times \ldots \times \tau_k \to \tau_{k+1}$, $h \in \{1, \ldots, k+1\}$, *and* $\pi \in \mathit{Pos}(\tau_h)$

Definition 2.4 (Type components)**.** *For a type $\tau \in \mathit{Types}(\Omega, \mathcal{W})$ and a type position $\pi \in \mathit{Pos}(\tau)$, $\tau|_\pi \in \mathit{Types}(\Omega, \mathcal{W})$ denotes the* type component at position π in τ*:*

$$\tau|_\epsilon := \tau$$
$$\mathit{str}[\tau_1, \ldots, \tau_k]|_{h\pi} := \tau_h|_\pi$$
$$\big(\tau_1 \times \ldots \times \tau_k \to \tau_{k+1}\big)|_{h\pi} := \tau_h|_\pi$$

[2] The empty sequence is written as ϵ, and $\mathbf{123} \in \mathbb{N}^*$ denotes the sequence of 1, 2, and 3 to distinguish it from the natural number 123. (There are no positions greater than 9 in our examples.)

Definition 2.5 (Type symbols)**.** *A* type symbol *is a type variable* $@A$*, a type constructor str, or the function type symbol "*$\rightarrow$*".*

For a type $\tau \in Types(\Omega, \mathcal{W})$ *and a type position* $\pi \in Pos(\tau)$*,* $\tau\|_\pi$ *denotes the* type symbol at position π in τ*:*

$$\tau\|_\pi := \begin{cases} @A & \text{if } \tau|_\pi = @A \\ str & \text{if } \tau|_\pi = str[\tau_1, \ldots, \tau_k] \\ \rightarrow & \text{if } \tau|_\pi = \tau_1 \times \ldots \times \tau_k \rightarrow \tau_{k+1} \end{cases} .$$

Example 2.6.

- For $\tau := \mathbb{N}$, $Pos(\tau) = \{\epsilon\}$, and $\tau|_\epsilon = \mathbb{N}$.
- For $\tau := list[\mathbb{N}]$, $Pos(\tau) = \{\epsilon, \mathbf{1}\}$, $\tau|_\epsilon = list[\mathbb{N}]$, $\tau\|_\epsilon = list$, and $\tau|_{\mathbf{1}} = \mathbb{N}$.
- For $\tau := @A \rightarrow pair[\mathbb{N}, @A]$:
  - $Pos(\tau) = \{\epsilon, \mathbf{1}, \mathbf{2}, \mathbf{21}, \mathbf{22}\}$
  - $\tau|_\epsilon = @A \rightarrow pair[\mathbb{N}, @A]$ and $\tau\|_\epsilon = \rightarrow$
  - $\tau|_{\mathbf{1}} = @A$ and $\tau\|_{\mathbf{1}} = @A$
  - $\tau|_{\mathbf{2}} = pair[\mathbb{N}, @A]$ and $\tau\|_{\mathbf{2}} = pair$
  - $\tau|_{\mathbf{21}} = \mathbb{N}$ and $\tau\|_{\mathbf{21}} = \mathbb{N}$
  - $\tau|_{\mathbf{22}} = @A$ and $\tau\|_{\mathbf{22}} = @A$ $\diamondsuit$

If a type τ contains a type variable $@A$, all occurrences of $@A$ in τ can be replaced with another type $\tau' \neq bool$. Note that Definition 2.1 classifies type variables as base types. Consequently, the following definition of type substitutions allows to substitute a type variable with a base type only. Thus the order of a type does not change by applying a substitution to it.

Definition 2.7 (Type substitutions)**.** *A* type substitution *is a mapping* $\theta : \mathcal{W} \rightarrow Types(\Omega, \mathcal{W})_0$ *with* $\theta(@A) \neq @A$ *for only finitely many* $@A \in \mathcal{W}$ *and* $\theta(@A) \neq bool$ *for all* $@A \in \mathcal{W}$*. As usual, we use the notation* $\{@A_1/\tau_1, \ldots, @A_k/\tau_k\}$ *for a type substitution* θ *with* $\theta(@A_h) = \tau_h$ *for all* $h = 1, \ldots, k$*.*

The homomorphic extension $\widehat{\theta}$ *of a type substitution* θ *to types is defined by:*

$$\widehat{\theta}(@A) := \theta(@A) \text{ for all } @A \in \mathcal{W}$$
$$\widehat{\theta}(str[\tau_1, \ldots, \tau_k]) := str[\widehat{\theta}(\tau_1), \ldots, \widehat{\theta}(\tau_k)]$$
$$\widehat{\theta}(\tau_1 \times \ldots \times \tau_k \rightarrow \tau_{k+1}) := \widehat{\theta}(\tau_1) \times \ldots \times \widehat{\theta}(\tau_k) \rightarrow \widehat{\theta}(\tau_{k+1})$$

We identify θ *with its homomorphic extension* $\widehat{\theta}$ *and usually just write* θ*.*

Definition 2.8 (Grounding type substitutions)**.** *A type substitution θ is a* grounding type substitution for $\tau \in \mathit{Types}(\Omega, \mathcal{W})$ *if $\theta(@A) \in \mathit{Types}(\Omega)$ for all type variables $@A$ occurring in τ. We let $\mathit{GndSubst}_\Omega(\tau)$ denote the set of grounding type substitutions for τ. We extend this notation to lists of types by $\mathit{GndSubst}_\Omega(\tau_1, \ldots, \tau_k) := \bigcap_{h=1}^{k} \mathit{GndSubst}_\Omega(\tau_h)$.*

Each $\mathcal{L}$-program P contains the predefined data structure $\mathbb{N}$ (i. e., a type constructor of arity 0, cf. Figure 2.1 on p. 31), so $\mathit{GndSubst}_{\Omega(P)}(\tau) \neq \emptyset$ for all $\tau \in \mathit{Types}(\Omega(P), \mathcal{W})$.

Example 2.9. For $\theta := \{@A/\mathbb{N}, @B/\mathit{list}[\mathbb{N}]\} \in \mathit{GndSubst}_\Omega(\mathit{pair}[@A, @B])$ we get:

- $\theta(@A) = \mathbb{N}$ and $\theta(@B) = \mathit{list}[\mathbb{N}]$
- $\theta(\mathit{pair}[@A, @B]) = \mathit{pair}[\mathbb{N}, \mathit{list}[\mathbb{N}]]$
- $\theta(\mathit{pair}[@A, @B] \to @B) = \mathit{pair}[\mathbb{N}, \mathit{list}[\mathbb{N}]] \to \mathit{list}[\mathbb{N}]$ ♢

2.1.2 Terms

We assume a family $\mathcal{V} = (\mathcal{V}_\tau)_{\tau \in \mathit{Types}(\Omega,\mathcal{W})}$ of countable sets $\mathcal{V}_\tau$ of *term variables of type* τ that are pairwise disjoint. Term variables of type *bool* are not allowed: $\mathcal{V}_{\mathit{bool}} = \emptyset$. A *term signature* is a family $\Sigma = (\Sigma_\tau)_{\tau \in \mathit{Types}(\Omega,\mathcal{W})}$ of pairwise disjoint sets of symbols that denote *function symbols*. We use the common notation $x : \tau$ and $f : \tau$ to denote a term variable $x \in \mathcal{V}_\tau$ and a function symbol $f \in \Sigma_\tau$ if $\mathcal{V}$ and Σ are obvious from the context. For simplicity, we sometimes use set notation $\mathcal{V} = \{x_1, \ldots, x_n\}$ to denote a family $(\mathcal{V}_\tau)_{\tau \in \mathit{Types}(\Omega,\mathcal{W})}$ where $\mathcal{V}_\tau$ contains those term variables $x \in \mathcal{V}$ that have type τ. The *order* of a term variable or a function symbol is the order of its type.

Initially, the term signature contains several "built-in" function symbols. The initial type and term signatures can be extended by data structure and procedure definitions, see Sections 2.1.3 and 2.1.4.

Definition 2.10 (Initial signature)**.** *The* initial type signature Ω_{init} *is defined by $\Omega_{\mathit{init},0} := \{\mathit{bool}\}$ and $\Omega_{\mathit{init},k} := \emptyset$ for all $k \in \mathbb{N} \setminus \{0\}$. The* initial term signature Σ_{init} *consists of the following function symbols:*

- $\mathit{true} : \mathit{bool}$
- $\mathit{false} : \mathit{bool}$
- $= : @A \times @A \to \mathit{bool}$
- $\mathit{if}_{\mathit{bool}} : \mathit{bool} \times \mathit{bool} \times \mathit{bool} \to \mathit{bool}$
- $\mathit{if}_{@A} : \mathit{bool} \times @A \times @A \to @A$

Since we disallow the substitution of a type variable $@A$ with *bool*, the initial term signature Σ_{init} defines a function symbol if_{bool} for conditional expressions of type *bool* and a function symbol $if_{@A}$ for conditional expressions of types $@A \neq bool$. Function symbol if_{bool} is used to form the usual logical connectives $\neg$, $\wedge$, $\vee$, and $\rightarrow$, see Section 2.1.5. As it is always clear from the context which *if*-symbol is meant, we omit the type index from now on.

For a term signature $\Sigma(P)$, we write $\Sigma^{\mathsf{cond}} \subset \Sigma$ for the sub-signature of *conditional function symbols*. Σ^{cond} contains if_{bool} and $if_{@A}$ as well as *case*-symbols, cf. Definition 2.32 (p. 31).

Each term variable and each function symbol is a term. If a term has a function type, it can be applied to other terms. A term can also be a *let*-expression $let\{x := t_1;\ t_0\}$: Each occurrence of x in t_0 stands for term t_1. Terms of a function type can be constructed by λ-abstraction: $\lambda x_1 : \tau_1, \ldots, x_n : \tau_n . t$ denotes a function with n parameters $x_1, \ldots, x_n$. In this λ-expression, term t represents the "return value" of this function.

Formally, terms are defined as follows:

Definition 2.11 (Terms)**.** *The family* $\mathcal{T}(\Sigma, \mathcal{V}) = (\mathcal{T}(\Sigma, \mathcal{V})_\tau)_{\tau \in Types(\Omega, \mathcal{W})}$ *of* terms *over a type signature* Ω*, a term signature* Σ*, and a family* $\mathcal{V}$ *of term variables is the smallest family of sets that satisfies the following requirements:*

- $\mathcal{V}_\tau \subseteq \mathcal{T}(\Sigma, \mathcal{V})_\tau$
- $f \in \mathcal{T}(\Sigma, \mathcal{V})_{\theta(\tau)}$ *for all* $f \in \Sigma_\tau$ *and all type substitutions* θ
- $t_0(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$ *if* $t_0 \in \mathcal{T}(\Sigma, \mathcal{V})_{\tau_1 \times \ldots \times \tau_n \rightarrow \tau}$*,* $n \geq 1$*, and* $t_i \in \mathcal{T}(\Sigma, \mathcal{V})_{\tau_i}$ *for all* $i = 1, \ldots, n$
- $let\{x := t_1;\ t_0\} \in \mathcal{T}(\Sigma, \mathcal{V})_{\tau_0}$ *if* $x \notin \mathcal{V}$*,* $t_1 \in \mathcal{T}(\Sigma, \mathcal{V})_{\tau_1}$ *for some type* $\tau_1 \in Types(\Omega, \mathcal{W})_0$*, and* $t_0 \in \mathcal{T}(\Sigma, \mathcal{V} \cup \{x : \tau_1\})_{\tau_0}$ *for some* $\tau_0 \in Types(\Omega, \mathcal{W})_0$
- $\big(\lambda x_1 : \tau_1, \ldots, x_n : \tau_n . t\big) \in \mathcal{T}(\Sigma, \mathcal{V})_{\tau_1 \times \ldots \times \tau_n \rightarrow \tau}$ *if* $n \geq 1$*,* $x_i \notin \mathcal{V}$ *for all* $i = 1, \ldots, n$*, and* $t \in \mathcal{T}(\Sigma, \mathcal{V} \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\})_\tau$

A ground term *is a term* $t \in \mathcal{T}(\Sigma, \emptyset)_\tau$ *for some* $\tau \in Types(\Omega)$*; i. e.,* τ *is a ground type and* t *does not contain term variables. We write* $\mathcal{T}(\Sigma)$ *for the family of ground terms. If the type* τ *of a term* $t \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$ *does not matter or is obvious from the context, we simply write* $t \in \mathcal{T}(\Sigma, \mathcal{V})$ *or* $t \in \mathcal{T}(\Sigma)$*. The* order *of a term* $t \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$ *is the order of type* τ*.*

Actually, $\mathcal{T}(\Sigma, \mathcal{V})$ should also be parameterized by the type signature Ω. Since the type signature $\Omega(P)$ of an $\mathcal{L}$-program P is implicitly defined by the term signature $\Sigma(P)$ (cf. Section 2.1.3), we omit the reference to Ω to keep the notation readable.

A *let*-expression $let\{x := t_1;\ t_0\}$ for a term variable $x \in \mathcal{V}_{\tau'}$ could be regarded as shorthand notation for $(\lambda x:\tau'.\, t_0)(t_1)$. Although these terms are semantically equivalent, we consider them as syntactically different. This allows us to treat them differently in proofs: A λ-expression $(\lambda x:\tau'.\, t_0)(t_1)$ is eagerly β-reduced to $t_0[x/t_1]$ (i. e., all occurrences of x in t_0 are substituted by t_1, see Definition 2.19 below), whereas a *let*-expression $let\{x := t_1;\ t_0\}$ remains a *let*-expression until it seems beneficial to "β-reduce" it to $t_0[x/t_1]$, see [73].

Example 2.12. Let $\mathcal{V}$ contain $x, y \in \mathcal{V}_{\mathbb{N}}$, $k \in \mathcal{V}_{list[\mathbb{N}]}$, and $p \in \mathcal{V}_{@A\to bool}$. Let Σ contain

- $dbl \in \Sigma_{\mathbb{N}\to\mathbb{N}}$,
- $every \in \Sigma_{(@A\to bool)\times list[@A]\to bool}$, and
- $even \in \Sigma_{\mathbb{N}\to bool}$.

Then:

- $x = y \in \mathcal{T}(\Sigma,\mathcal{V})_{bool}$
- $every \in \mathcal{T}(\Sigma,\mathcal{V})_{(@A\to bool)\times list[@A]\to bool}$
- $every \in \mathcal{T}(\Sigma,\mathcal{V})_{(\mathbb{N}\to bool)\times list[\mathbb{N}]\to bool}$
- $every(even, k) \in \mathcal{T}(\Sigma,\mathcal{V})_{bool}$
- $let\{z := dbl(x);\ even(z)\} \in \mathcal{T}(\Sigma,\mathcal{V})_{bool}$
- $\lambda l : list[@A].\, every(p, l) \in \mathcal{T}(\Sigma,\mathcal{V})_{list[@A]\to bool}$ ♢

Terms of type $\tau_1 \times \ldots \times \tau_k \to bool$ for some $k \in \mathbb{N}$ denote *predicates*. For example, if *even* is a function symbol of type $\mathbb{N} \to bool$, then *even* (considered as a term) is a predicate. Furthermore, if $n:\mathbb{N}$, then $even(n)$ is also a predicate. This predicate can be transformed back into a predicate of type $\mathbb{N} \to bool$ by encapsulating it into a λ-expression: $\lambda n:\mathbb{N}.\, even(n)$.

We say that term variable n occurs *free* in $even(n)$, whereas it does *not* occur free in $\lambda n:\mathbb{N}.\, even(n)$.

Definition 2.13 (Free term variables)**.** *Let $t \in \mathcal{T}(\Sigma,\mathcal{V})$ be a term. The set $\mathcal{V}_{\mathsf{f}}(t) \subseteq \mathcal{V}$ of* free term variables in t *is defined by:*

$$\begin{aligned}
&\mathcal{V}_{\mathsf{f}}(x) := \{x\} \text{ for all } x \in \mathcal{V}\\
&\mathcal{V}_{\mathsf{f}}(f) := \emptyset \text{ for all } f \in \Sigma\\
&\mathcal{V}_{\mathsf{f}}(t_0(t_1,\ldots,t_n)) := \mathcal{V}_{\mathsf{f}}(t_0) \cup \mathcal{V}_{\mathsf{f}}(t_1) \cup \ldots \cup \mathcal{V}_{\mathsf{f}}(t_n)\\
&\mathcal{V}_{\mathsf{f}}(let\{x := t_1;\ t_0\}) := (\mathcal{V}_{\mathsf{f}}(t_0) \setminus \{x\}) \cup \mathcal{V}_{\mathsf{f}}(t_1)\\
&\mathcal{V}_{\mathsf{f}}(\lambda x_1:\tau_1,\ldots,x_n:\tau_n.\, t) := \mathcal{V}_{\mathsf{f}}(t) \setminus \{x_1,\ldots,x_n\}
\end{aligned}$$

A term t is closed *iff $\mathcal{V}_{\mathsf{f}}(t) = \emptyset$.*

For a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$, we sometimes need to know which function symbols occur in t. Therefore the following definition defines the signature $\Sigma(t) \subseteq \Sigma$ of term t that comprises all function symbols that occur in t. Similarly to the definition of free variables, we also define the signature $\Sigma_{\mathsf{f}}(t) \subseteq \Sigma(t)$ of all function symbols that occur outside of λ-expressions in t.

Definition 2.14 (Signature of a term)**.** *For a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$, we define $\Sigma(t)$ as the $\subseteq$-minimal subset Σ' of Σ such that $t \in \mathcal{T}(\Sigma', \mathcal{V})$. The signature $\Sigma_{\mathsf{f}}(t) \subseteq \Sigma(t)$ of function symbols outside of λ-expressions in t is defined by:*

$$\Sigma_{\mathsf{f}}(x) := \emptyset \text{ for all } x \in \mathcal{V}$$
$$\Sigma_{\mathsf{f}}(f) := \{f\} \text{ for all } f \in \Sigma$$
$$\Sigma_{\mathsf{f}}(t_0(t_1, \ldots, t_n)) := \Sigma_{\mathsf{f}}(t_0) \cup \Sigma_{\mathsf{f}}(t_1) \cup \ldots \cup \Sigma_{\mathsf{f}}(t_n)$$
$$\Sigma_{\mathsf{f}}(\mathit{let}\{x := t_1;\ t_0\}) := \Sigma_{\mathsf{f}}(t_0) \cup \Sigma_{\mathsf{f}}(t_1)$$
$$\Sigma_{\mathsf{f}}(\lambda x_1 : \tau_1, \ldots, x_n : \tau_n.\, t) := \emptyset$$

Example 2.15. Let Σ be as in Example 2.12 with $\mathit{rev} \in \Sigma_{\mathit{list}[@A] \to \mathit{list}[@A]}$.

$$\Sigma\big(\mathit{every}(\lambda n : \mathbb{N}.\, \mathit{dbl}(n) = n,\ \mathit{rev}(k))\big) = \{\mathit{every}, \mathit{dbl}, =, \mathit{rev}\}$$
$$\Sigma_{\mathsf{f}}\big(\mathit{every}(\lambda n : \mathbb{N}.\, \mathit{dbl}(n) = n,\ \mathit{rev}(k))\big) = \{\mathit{every}, \mathit{rev}\} \qquad \diamondsuit$$

According to Definition 2.11, a term t can be composed of other terms. These terms occur at certain positions:

Definition 2.16 (Term positions)**.** *For a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$, $\mathit{Pos}(t) \subset \mathbb{N}^*$ denotes the set of all* term positions *of t. It is defined as the smallest set that satisfies the following requirements:*

1. $\epsilon \in \mathit{Pos}(t)$
2. $i\pi \in \mathit{Pos}(t)$ *if* $t = t_0(t_1, \ldots, t_n)$, $i \in \{0, \ldots, n\}$, *and* $\pi \in \mathit{Pos}(t_i)$
3. $i\pi \in \mathit{Pos}(t)$ *if* $t = \mathit{let}\{x := t_1;\ t_0\}$, $i \in \{0, 1\}$, *and* $\pi \in \mathit{Pos}(t_i)$
4. $\mathbf{0}\pi \in \mathit{Pos}(t)$ *if* $t = \lambda x_1 : \tau_1, \ldots, x_n : \tau_n.\, t$ *and* $\pi \in \mathit{Pos}(t)$

The subset $\mathit{TLPos}(t) \subseteq \mathit{Pos}(t)$ of top-level term positions *of t is defined as the smallest set that satisfies requirements (1)–(3).*

The set of term positions thus consists of all top-level positions and additionally contains positions within λ-expressions. Each term position is the address of a *subterm*:

Definition 2.17 (Subterms)**.** *For a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ and a term position $\pi \in \mathit{Pos}(t)$, $t|_\pi$ denotes the* subterm at position π in t*:*

$$t|_\epsilon := t$$
$$t_0(t_1, \ldots, t_n)|_{i\pi} := t_i|_\pi$$
$$\mathit{let}\{x := t_1;\ t_0\}|_{i\pi} := t_i|_\pi$$
$$\big(\lambda x_1 : \tau_1, \ldots, x_n : \tau_n.\, t\big)|_{\mathbf{0}\pi} := t|_\pi$$

A term $s \in \mathcal{T}(\Sigma, \mathcal{V})$ is called a subterm of $t \in \mathcal{T}(\Sigma, \mathcal{V})$*, written $s \leq_{\mathcal{T}} t$, iff there exists a term position $\pi \in Pos(t)$ such that $s = t|_\pi$. We call s a* proper subterm of t*, written $s <_{\mathcal{T}} t$, iff there exists a term position $\pi \in Pos(t) \setminus \{\epsilon\}$ such that $s = t|_\pi$.*

We write $t[\pi \leftarrow s]$ for the replacement of the subterm at position π in t with s*:*

$$
\begin{aligned}
&t[\epsilon \leftarrow s] := s \\
&t_0(t_1, \ldots, t_n)[\mathbf{0}\pi \leftarrow s] := t_0[\pi \leftarrow s](t_1, \ldots, t_n) \\
&t_0(t_1, \ldots, t_n)[i\pi \leftarrow s] := t_0(t_1, \ldots, t_{i-1}, t_i[\pi \leftarrow s], t_{i+1}, \ldots, t_n)\ ;\ i \geq 1 \\
&\mathit{let}\{x := t_1;\ t_0\}[\mathbf{1}\pi \leftarrow s] := \mathit{let}\{x := t_1[\pi \leftarrow s];\ t_0\} \\
&\mathit{let}\{x := t_1;\ t_0\}[\mathbf{0}\pi \leftarrow s] := \mathit{let}\{x := t_1;\ t_0[\pi \leftarrow s]\} \\
&\big(\lambda x_1 : \tau_1, \ldots, x_n : \tau_n.\, t\big)[\mathbf{0}\pi \leftarrow s] := \lambda x_1 : \tau_1, \ldots, x_n : \tau_n.\, t[\pi \leftarrow s]
\end{aligned}
$$

A term variable x that occurs free in s may be "accidentally bound" in $t[\pi \leftarrow s]$; e. g., $\big(\lambda x, y.\, f(x, x)\big)[\mathbf{02} \leftarrow y] = \lambda x, y.\, f(x, y)$. One can either avoid this by renaming the bound variable y in $\lambda x, y.\, f(x, x)$ to z so that y remains free or by using $t[\pi \leftarrow s]$ only if all occurrences of the variables in $\mathcal{V}_{\mathsf{f}}(s)$ are free in $t|_\pi$. We use the latter alternative; i. e., s always contains only free variables that occurred free at term position π before.

Example 2.18. For term $t := \mathit{map}(\lambda n : \mathbb{N}.\, n + m,\ k)$ we get

$$
\begin{aligned}
\mathit{TLPos}(t) &= \{\epsilon, \mathbf{0}, \mathbf{1}, \mathbf{2}\} \\
\mathit{Pos}(t) &= \mathit{TLPos}(t) \cup \{\mathbf{10}, \mathbf{100}, \mathbf{101}, \mathbf{102}\}
\end{aligned}
$$

with

$$
\begin{aligned}
t|_{\mathbf{0}} &= \mathit{map} \\
t|_{\mathbf{1}} &= \lambda n : \mathbb{N}.\, n + m \\
t|_{\mathbf{2}} &= k \\
t|_{\mathbf{10}} &= n + m \\
t|_{\mathbf{100}} &= + \\
t|_{\mathbf{101}} &= n \\
t|_{\mathbf{102}} &= m\,.
\end{aligned}
$$

For instance, $t[\mathbf{102} \leftarrow x] = \mathit{map}(\lambda n : \mathbb{N}.\, n + x,\ k)$. ♢

Similarly to type substitutions, a term substitution replaces all occurrences of a term variable $x : \tau$ in some term t' (or finitely many term variables in general) with a term t of the same type τ. Again, we need to make sure that the free variables in t remain free when an occurrence of x in t' is replaced with t.

Definition 2.19 (Term substitutions)**.** *A* term substitution *is a mapping* $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$ *such that*

1. *the* domain $dom(\sigma) := \{ x \in \mathcal{V} \mid \sigma(x) \neq x \}$ *is finite and*
2. *for all* $\tau \in Types(\Omega, \mathcal{W})$ *and all* $x \in \mathcal{V}_\tau$, $\sigma(x) \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$.

As usual, we use the notation $\{x_1/t_1, \ldots, x_n/t_n\}$ *for a term substitution* σ *with* $\sigma(x_i) := t_i$ *for all* $i = 1, \ldots, n$.

The restriction $\sigma|_V$ *of* σ *to some set* $V \subseteq \mathcal{V}$ *of term variables is defined by*

$$\sigma|_V(x) := \begin{cases} \sigma(x) & \text{if } x \in V \\ x & \text{otherwise.} \end{cases}$$

Let $t \in \mathcal{T}(\Sigma, \mathcal{V})$ *be a term. A term substitution* σ *is* applicable to t *iff the following requirements are satisfied:*

3. *For each subterm* $let\{x := t_1;\ t_0\}$ *of* t *and all variables* $y \in \mathcal{V}_\mathsf{f}(t_0) \setminus \{x\}$, $x \notin \mathcal{V}_\mathsf{f}(\sigma(y))$.
4. *For each subterm* $\lambda x_1 : \tau_1, \ldots, x_n : \tau_n . t'$ *of* t *and all variables* $y \in \mathcal{V}_\mathsf{f}(t') \setminus \{x_1, \ldots, x_n\}$, $x_1, \ldots, x_n \notin \mathcal{V}_\mathsf{f}(\sigma(y))$.

The homomorphic extension $\widehat{\sigma}$ *of a term substitution* σ *to terms is defined by the following equations. If* σ *is not applicable to a given term* t*, we assume that* t *is implicitly transformed into a term* t' *that* σ *is applicable to by renaming the bound variables in* t.

$$\begin{aligned}
&\widehat{\sigma}(x) := \sigma(x) \text{ for all } x \in \mathcal{V} \\
&\widehat{\sigma}(f) := f \text{ for all } f \in \Sigma \\
&\widehat{\sigma}(t_0(t_1, \ldots, t_n)) := \widehat{\sigma}(t_0)\big(\widehat{\sigma}(t_1), \ldots, \widehat{\sigma}(t_n)\big) \\
&\widehat{\sigma}(let\{x := t_1;\ t_0\}) := let\{x := \widehat{\sigma}(t_1);\ \widehat{\sigma}|_{\mathcal{V}\setminus\{x\}}(t_0)\} \\
&\widehat{\sigma}(\lambda x_1 : \tau_1, \ldots, x_n : \tau_n . t) := \lambda x_1 : \tau_1, \ldots, x_n : \tau_n . \widehat{\sigma}|_{\mathcal{V}\setminus\{x_1,\ldots,x_n\}}(t)
\end{aligned}$$

We identify σ *with its homomorphic extension* $\widehat{\sigma}$ *and usually just write* σ. *The notation* $t[x_1/t_1, \ldots, x_n/t_n]$ *abbreviates* $\widehat{\sigma}(t)$ *for* $\sigma := \{x_1/t_1, \ldots, x_n/t_n\}$. *For a term* t *with* $\mathcal{V}_\mathsf{f}(t) \supseteq \{x_1, \ldots, x_n\}$ *and terms* $q_1, \ldots, q_n$ *we write* $t[\vec{q}]$ *for* $t[x_1/q_1, \ldots, x_n/q_n]$.

Example 2.20. Term substitution $\sigma := \{m/dbl(n), k/k'\}$ is not applicable to term $t := map(\lambda n : \mathbb{N}.\, n + m,\ k)$, because the occurrence of n in $dbl(n)$ would be bound by the surrounding λ-expression. However, we can transform t into $t' := map(\lambda x : \mathbb{N}.\, x + m,\ k)$ that σ is applicable to: $\sigma(t') = map(\lambda x : \mathbb{N}.\, x + dbl(n),\ k')$. ♢

Definition 2.21 (β-normal)**.** *A term* $(\lambda x_1 : \tau_1, \ldots, x_n : \tau_n.\, t)(t_1, \ldots, t_n)$ *is called a* β-redex*. Its* β-reduct *is defined as* $t[x_1/t_1, \ldots, x_n/t_n]$*. We say that a term* $t \in \mathcal{T}(\Sigma, \mathcal{V})$ *is* β-normal *iff it does not contain a* β*-redex as subterm.*

Example 2.22. Term $(\lambda n : \mathbb{N}.\, dbl(n))(m)$ is a β-redex. Its β-reduct is $dbl(m)$. Thus $x + (\lambda n : \mathbb{N}.\, dbl(n))(m)$ is *not* β-normal, whereas $x + dbl(m)$ and $map(\lambda n : \mathbb{N}.\, dbl(n),\, k)$ are β-normal. ♢

Definition 2.23 (η-long)**.** *A term* $\lambda x_1 : \tau_1, \ldots, x_n : \tau_n.\, t(x_1, \ldots, x_n)$ *is called an* η-redex*. The* η-reduct *of this* η*-redex is defined as* t*. The* η-expansion *of a term* $t : \tau_1 \times \ldots \times \tau_n \to \tau$ *is defined as* $\lambda x_1 : \tau_1, \ldots, x_n : \tau_n.\, t(x_1, \ldots, x_n)$*. We write* $t =_\eta t'$ *iff* t *can be transformed into* t' *by finitely many* η*-expansions or* η*-reductions of subterms of* t *and* t'*.*

A term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ *is said to be* η-long *iff no subterm of* t *can be* η*-expanded without introducing a* β*-redex.*

Example 2.24. Term $\lambda n : \mathbb{N}.\, dbl(n)$ is an η-redex. Its η-reduct is dbl. Conversely, the η-expansion of dbl is $\lambda n : \mathbb{N}.\, dbl(n)$. We have $map(dbl, k) =_\eta map(\lambda n : \mathbb{N}.\, dbl(n),\, k)$.

Term $map(dbl, k)$ is *not* η-long, whereas $map(\lambda n : \mathbb{N}.\, dbl(n),\, k)$ is η-long, because η-expansion of map or dbl would introduce β-redexes. ♢

A term t may contain function calls inside and outside a λ-expression. Function calls inside a λ-expression are *indirect function calls*:

Definition 2.25 (Direct and indirect function calls)**.** *Let* $\pi \in Pos(t)$ *be a term position of some* η*-long term* $t \in \mathcal{T}(\Sigma, \mathcal{V})$ *with* $t|_\pi = t_0(t_1, \ldots, t_n)$ *for some terms* $t_0, \ldots, t_n$ *and* $n \geq 0$*. We say that* $t_0(t_1, \ldots, t_n)$ *is a* direct function call *iff* $\pi \in TLPos(t)$*. Otherwise, it is an* indirect function call*.*

Example 2.26. Term $t := map(\lambda n : \mathbb{N}.\, n + m,\, k)$ contains a *direct* function call $map(\ldots)$ and an *indirect* function call $n + m$. ♢

Terms that are both β-normal and η-long are usually called *canonical*. To simplify the analysis of terms, it is convenient to assume furthermore that a term does not contain a conditional expression within the condition of another conditional expression. Also, conditional expressions should not occur within the arguments of a function call $t_0(t_1, \ldots, t_n)$. We call such terms *normalized* [89]:

Definition 2.27 (Normalized terms)**.** *A term* $t \in \mathcal{T}(\Sigma, \mathcal{V})$ *is* normalized *iff the following requirements are satisfied:*

- t *is* β*-normal and* η*-long.*
- *For each subterm if*$\{b, t_1, t_2\}$ *of* t*,* $\Sigma_{\mathsf{f}}(b) \cap \Sigma^{\mathsf{cond}} = \emptyset$*.*
- *For each subterm case*$\{t'; cons_1 : t_1, \ldots, cons_m : t_m\}$ *of* t*,* $\Sigma_{\mathsf{f}}(t') \cap \Sigma^{\mathsf{cond}} = \emptyset$*.*

- *For each subterm* $t_0(t_1, \ldots, t_n)$ *of* t *with* $t_0 \notin \Sigma^{\mathsf{cond}}$, $\Sigma_{\mathsf{f}}(t_i) \cap \Sigma^{\mathsf{cond}} = \emptyset$ *for all* $i = 1, \ldots, n$.
- *For each subterm* $let\{x := t_1;\ t_0\}$ *of* t, $\Sigma_{\mathsf{f}}(t_1) \cap \Sigma^{\mathsf{cond}} = \emptyset$.

Example 2.28.

- $map(f, k)$ is not normalized, because it is not η-long.
- $map(\lambda x : @A.\ f(x),\ k)$ is normalized.
- $if\{if\{x > y, true, y > x\}, if\{x = y, false, true\}, true\}$ is not normalized, because an *if*-condition contains a conditional expression.
- $if\{x > y, if\{x = y, false, true\}, if\{y > x, if\{x = y, false, true\}, true\}\}$ is a normalized term.
- $let\{x := if\{?0(z), 0, {}^{-}(z)\};\ x + y\}$ is not normalized. ♢

For each $t \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$, we can obtain a normalized term $Normalize(t) \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$ that is semantically equivalent to t by successively applying the following transformations until no transformation is applicable (in which case the term is normalized):

- Reduce each β-redex in t to its β-reduct.
- Replace each subterm $t' : \tau_1 \times \ldots \times \tau_n \to \tau$ of t with its η-expansion if this does not introduce a β-redex.
- Replace each subterm $cond\{cond'\{t';\ t'_1, \ldots, t'_n\};\ t_1, \ldots, t_m\}$ of t with

  $$cond'\{t';\ cond\{t'_1;\ t_1, \ldots, t_m\}, \ldots, cond\{t'_n;\ t_1, \ldots, t_m\}\}\ ,$$

  where $cond, cond' \in \Sigma^{\mathsf{cond}}$.
- Replace each subterm $t_0(t_1, \ldots, t_{i-1}, cond\{t';\ t'_1, \ldots, t'_m\}, t_{i+1}, \ldots, t_n)$ of t with $cond\{t';\ T_1, \ldots, T_m\}$, where $t_0 \notin \Sigma^{\mathsf{cond}}$, $cond \in \Sigma^{\mathsf{cond}}$, and $T_j := t_0(t_1, \ldots, t_{i-1}, t'_j, t_{i+1}, \ldots, t_n)$ for $j = 1, \ldots, m$.
- Replace each subterm $let\{x := cond\{t';\ t_1, \ldots, t_m\};\ t_0\}$ of t with $cond\{t';\ let\{x := t_1;\ t_0\}, \ldots, let\{x := t_m;\ t_0\}\}$, where $cond \in \Sigma^{\mathsf{cond}}$.

Each function symbol f is associated with a so-called *context requirement* c_f [72], which is a Boolean term:

Definition 2.29 (Context requirement of a function symbol)**.** *Let* $f : \tau_1 \times \ldots \times \tau_n \to \tau$ *be a function symbol with formal parameters* $x_1 : \tau_1, \ldots, x_n : \tau_n$. *The* context requirement c_f of f *is a term* $c_f \in \mathcal{T}(\Sigma \setminus \{f\}, \{x_1, \ldots, x_n\})_{bool}$.

The context requirement stipulates a precondition that restricts the arguments that f may be applied to: Function f may only be applied to $x_1, \ldots, x_n$ if $c_f[x_1, \ldots, x_n]$ holds. For procedures *proc*, c_{proc} is part of the procedure definition. For other function symbols f, we define the context requirement c_f along with the signature of f. If we do not explicitly mention the context requirement of a function symbol f, then $c_f := \mathit{true}$.

Example 2.30. A reasonable context requirement for selector $^-(x:\mathbb{N})$ is $?^+(x)$ (cf. Definition 2.32 on p. 31), because x should not be `0` when the predecessor of x is to be computed. Context requirement of constructor $^+(x:\mathbb{N})$ is *true*, as the successor of x can be computed for any natural number x. $\diamondsuit$

Pretty Printing. We usually write normalized terms in the so-called η-*short form* (which is obtained by η-reducing each η-redex). For instance, the body of procedure *groundterm* (cf. Figure 1.5 on p. 9) contains the subterm $\mathit{every}(\mathit{groundterm}, \mathit{args}(t))$. While this notation is used for pretty printing and may also be employed by the user, it actually abbreviates the normalized term

$$\mathit{every}(\lambda s : \mathit{term}[@V, @F].\ \mathit{groundterm}(s),\ \mathit{args}(t))\,.$$

Furthermore, each function symbol can be assigned a *fixity* [90]:

`[prefix]` Function calls are written as $f(\ldots)$, i. e., the function symbol precedes the list of arguments of the function call. The formal treatment (as in Definition 2.11) always assumes prefix notation.

Example: $hd(k)$.

`[postfix]` Function calls are written as $(\ldots)f$, i. e., the function symbol follows after the list of arguments of the function call.

Example: $(n)!$ for the factorial function.

`[outfix]` Function calls are written as $f \ldots f$, i. e., the function symbol surrounds the list of arguments of the function call.

Example: $|\,k\,|$ for the length of list k.

`[infix]` Function calls are written as (xfy), i. e., the function symbol stands between the two arguments of the function call. The whole function call is surrounded by parentheses.

Example: $(x \bullet y)$ for the pair of x and y (cf. Figure 2.1 on p. 31).

`[infixl,`N`]` Function calls are written as xfy, i. e., the "left-associative" function symbol stands between the two arguments of the function call. A term $x\,f\,y\,f\,z$ is understood as $(x\,f\,y)\,f\,z$. N is a natural number to denote the *precedence* of the function symbol. If the precedence of f

is higher than the precedence of g, function call $x\ f\ y\ g\ z$ is understood as $(x\ f\ y)\ g\ z$; otherwise it is understood as $x\ f\ (y\ g\ z)$.

Example: $x - y$ (cf. Figure 3.3 on p. 70).

`[infixr, N]` Function calls are written as xfy, i. e., the "right-associative" function symbol stands between the two arguments of the function call. A term $x\ f\ y\ f\ z$ is understood as $x\ f\ (y\ f\ z)$. N denotes again the precedence of the function symbol.

Example: $x :: y :: \varepsilon$ (cf. Figure 2.1 on p. 31).

`[infix*, N]` Function calls are written as xfy, i. e., the "non-associative" function symbol stands between the two arguments of the function call. This infix notation should be used when the signature of f does not allow terms of the form $x\ f\ y\ f\ z$.

Example: $x > y$.

2.1.3 Data Structure Definitions

A data structure definition defines a new type constructor *str* as well as data constructors $cons_i$:

Definition 2.31 (Data structure definitions)**.** *An expression of the form*

$$\begin{array}{l}\texttt{structure}\ str[@A_1, \ldots, @A_k] <= \\ \quad cons_1(sel_{1,1} : \tau_{1,1}, \ldots, sel_{1,n_1} : \tau_{1,n_1}), \\ \quad \ldots, \\ \quad cons_m(sel_{m,1} : \tau_{m,1}, \ldots, sel_{m,n_m} : \tau_{1,n_m})\end{array}$$

for $k \in \mathbb{N}$ and pairwise different identifiers str, $cons_i$, $sel_{i,j}$, and $@A_h$ is called a data structure definition *for an $\mathcal{L}$-program P iff*

1. *$str \notin \Omega(P)_h$ for all $h \in \mathbb{N}$,*
2. *$m \geq 1$ and $cons_i \notin \Sigma(P)$ for all $i = 1, \ldots, m$,*
3. *$sel_{i,j} \notin \Sigma(P)$ for all $i = 1, \ldots, m$ and all $j = 1, \ldots, n_i$,*
4. *$\tau_{i,j} \in Types(\Omega(P'), \{@A_1, \ldots, @A_k\})_0$ for all $i = 1, \ldots, m$ and all $j = 1, \ldots, n_i$, where $\Omega(P')$ extends $\Omega(P)$ by $str \in \Omega(P')_k$,*
5. *each occurrence of $str[\ldots]$ in any of the types $\tau_{i,j}$ must be equal to $str[@A_1, \ldots, @A_k]$, and*
6. *there exists an $i \in \{1, \ldots, m\}$ such that type constructor str does not occur in $\tau_{i,1}, \ldots, \tau_{i,n_i}$.*

Data structure $str[@A_1, \ldots, @A_k]$ is called polymorphic *iff $k \geq 1$ and* monomorphic *otherwise.*

```
structure ℕ <=
  0,
  +(⁻ : ℕ)

structure list[@A] <=
  ε,
  [infixr, 100] ::(hd : @A, tl : list[@A])

structure pair[@A, @B] <=
  [infix] •(fst : @A, snd : @B)

structure term[@V, @F] <=
  var(vsym : @V),
  apply(fsym : @F, args : list[term[@V, @F]])
```

Figure 2.1: Data structure definitions $\mathbb{N}$, $list[@A]$, $pair[@A, @B]$, and $term[@V, @F]$

Requirements (1)–(3) demand that the identifiers str, $cons_i$, and $sel_{i,j}$ have not already been defined in program P. By (4), the $\tau_{i,j}$ are base types that may use $str, @A_1, \ldots, @A_k$, and previously defined type constructors. (5) prohibits an instantiation of $str[@A_1, \ldots, @A_k]$ in the $\tau_{i,j}$. (6) ensures that there exist "values" of type $str[@A_1, \ldots, @A_k]$ and is important for Definition 2.64 (p. 48).

Figure 2.1 shows some examples of data structure definitions.

Definition 2.32 (Signature of a data structure definition)**.** *An admissible data structure definition for an $\mathcal{L}$-program P extends the type signature $\Omega(P)$ to a type signature $\Omega(P') \supset \Omega(P)$ by type constructor $str \in \Omega(P')_k$. Furthermore, it extends the term signature $\Sigma(P)$ to a term signature $\Sigma(P') \supset \Sigma(P)$ by*

- $cons_i : \tau_{i,1} \times \ldots \times \tau_{i,n_i} \to \tau$*, called a* data constructor*,*
- $?cons_i : \tau \to bool$*, called a* structure predicate*,*
- $sel_{i,j} : \tau \to \tau_{i,j}$*, called a* selector*, with context requirement* $c_{sel_{i,j}} := ?cons_i(x)$*,*
- $case_{str,bool} : \tau \times \underbrace{bool \times \ldots \times bool}_{m \text{ times}} \to bool$
- $case_{str,@A} : \tau \times \underbrace{@A \times \ldots \times @A}_{m \text{ times}} \to @A$

where $\tau := str[@A_1, \ldots, @A_k]$.

For a term signature Σ, we define $\Sigma^{\mathsf{c}} \subset \Sigma$ as the signature of all data constructors in Σ, including true and false. Furthermore, $\Sigma^{\mathsf{cond}} \subset \Sigma$ is the signature of all conditional function symbols *in Σ; it comprises if_{bool}, $\mathit{if}_{@A}$, and all function symbols $\mathit{case}_{str,bool}$ and $\mathit{case}_{str,@A}$ of Σ.*

Example 2.33. Type constructor $\mathbb{N}$ denotes the data structure of natural numbers:

- $\mathsf{0} : \mathbb{N}$ is a data constructor.
- $^+ : \mathbb{N} \to \mathbb{N}$ is a data constructor and denotes the successor function.
- $?\mathsf{0} : \mathbb{N} \to \mathit{bool}$ returns *true* iff the argument is $\mathsf{0}$.
- $?^+ : \mathbb{N} \to \mathit{bool}$ returns *true* iff the argument is of the form $^+(\ldots)$, i. e., if it is different from $\mathsf{0}$.
- $^- : \mathbb{N} \to \mathbb{N}$ is the only selector of $^+(\ldots)$ and denotes the predecessor function. Its context requirement $c_{-(\ldots)} := ?^+(x)$ stipulates that it may only be applied to natural numbers different from $\mathsf{0}$.
- $\mathit{case}_{\mathbb{N},bool} : \mathbb{N} \times \mathit{bool} \times \mathit{bool} \to \mathit{bool}$ represents a case analysis over a natural number with branches of type *bool*. If the first argument is $\mathsf{0}$, it returns the second argument (of type *bool*). Otherwise it returns the third argument (of type *bool*).
- $\mathit{case}_{\mathbb{N},@A} : \mathbb{N} \times @A \times @A \to @A$ represents a case analysis over a natural number with branches of type $@A \neq \mathit{bool}$. If the first argument is $\mathsf{0}$, it returns the second argument (of type $@A$). Otherwise it returns the third argument (of type $@A$).

For a term $\underbrace{^+(\ldots{}^+}_{n \text{ times}}(\mathsf{0})\ldots)$ we often write n in examples. ♢

Example 2.34. $\mathit{list}[@A]$ denotes the data structure of polymorphic lists:

- $\varepsilon : \mathit{list}[@A]$ is a data constructor and denotes the empty list.
- $:: : @A \times \mathit{list}[@A] \to \mathit{list}[@A]$ is a data constructor and adds an item at the beginning of a list.
- $?\varepsilon : \mathit{list}[@A] \to \mathit{bool}$ returns *true* iff the argument is the empty list.
- $?:: : \mathit{list}[@A] \to \mathit{bool}$ returns *true* iff the argument is a non-empty list.
- $\mathit{hd} : \mathit{list}[@A] \to @A$ is a selector of $::$ with context requirement $c_{\mathit{hd}} := ?::(x)$. It returns the first item of a non-empty list.
- $\mathit{tl} : \mathit{list}[@A] \to \mathit{list}[@A]$ is a selector of $::$ with context requirement $c_{\mathit{tl}} := ?::(x)$. It returns the list without its first element.

- $case_{list,bool} : list[@A] \times bool \times bool \to bool$ represents a case analysis over a list with branches of type *bool*. If the first argument is an empty list, it returns the second argument (of type *bool*). Otherwise it returns the third argument (of type *bool*).

- $case_{list,@B} : list[@A] \times @B \times @B \to @B$ represents a case analysis over a list with branches of type $@B$. If the first argument is an empty list, it returns the second argument (of type $@B$). Otherwise it returns the third argument (of type $@B$). ♢

Example 2.35. $pair[@A, @B]$ denotes the data structure of polymorphic pairs:

- "$\bullet$" : $@A \times @B \to pair[@A, @B]$ is a data constructor and denotes the pair of two items.

- $?\bullet : pair[@A, @B] \to bool$ always returns *true*.

- $fst : pair[@A, @B] \to @A$ is a selector of $\bullet$ with context requirement $c_{fst} := ?\bullet(x)$. It returns the first item of a pair.

- $snd : pair[@A, @B] \to @B$ is a selector of $\bullet$ with context requirement $c_{snd} := ?\bullet(x)$. It returns the second item of a pair.

- $case_{pair,bool} : pair[@A, @B] \times bool \to bool$ represents a trivial case analysis over a pair with a branch of type *bool*. It returns the argument of type *bool*.

- $case_{pair,@C} : pair[@A, @B] \times @C \to @C$ represents a trivial case analysis over a pair with a branch of type $@C$. It returns the argument of type $@C$. ♢

Example 2.36. $term[@V, @F]$ denotes the data structure of terms over variables of type $@V$ and function symbols of type $@F$.

- $var : @V \to term[@V, @F]$ is a data constructor and represents a variable.

- $apply : @F \times list[term[@V, @F]] \to term[@V, @F]$ is a data constructor and represents the application of a function to a list of argument terms.

- $?var : term[@V, @F] \to bool$ returns *true* iff the argument represents a variable.

- $?apply : term[@V, @F] \to bool$ returns *true* iff the argument represents a function application.

- $vsym : term[@V, @F] \to @V$ is a selector of *var* with context requirement $c_{vsym} := ?var(x)$. It returns the variable symbol of a term that represents a variable.

- $fsym : term[@V, @F] \to @F$ is a selector of *apply* with context requirement $c_{fsym} := ?apply(x)$. It returns the leading function symbol of a term that represents a function application.

- $args : term[@V, @F] \to list[term[@V, @F]]$ is a selector of *apply* with context requirement $c_{args} := ?apply(x)$. Selector *args* returns the list $t_1 :: \ldots :: t_n :: \varepsilon$ of argument terms of a term $apply(f, t_1 :: \ldots :: t_n :: \varepsilon)$ that represents a function application.

- $case_{term,bool} : term[@V, @F] \times bool \times bool \to bool$ represents a case analysis over a term with branches of type *bool*. If the first argument represents a variable, it returns the second argument (of type *bool*). Otherwise it returns the third argument (of type *bool*).

- $case_{term,@A} : term[@V, @F] \times @A \times @A \to @A$ represents a case analysis over a term with branches of type $@A$. If the first argument represents a variable, it returns the second argument (of type $@A$). Otherwise it returns the third argument (of type $@A$). $\diamondsuit$

Similarly to function symbol *if*, we just write *case* and omit the type index, because it can always be reconstructed from the context.

Definition 2.37 (Occurrences of a type symbol)**.** *For a data constructor*

$$cons : \tau_1 \times \ldots \times \tau_n \to str[@A_1, \ldots, @A_k]$$

and a type symbol S, the set

$$Occ_S(cons) := \{ (j, \pi) \in \{1, \ldots, n\} \times \mathbb{N}^* \mid \pi \in Pos(\tau_j),\ \tau_j\|_\pi = S \}$$

contains the positions of all occurrences of S in the selector types of cons, given by a selector number j and a position π in τ_j. Data constructor cons is called reflexive *if $Occ_{str}(cons) \neq \emptyset$, and* irreflexive *otherwise. We write $\mathcal{C}^{\mathsf{refl}}_{str}$ for the set of all reflexive data constructors of type constructor str and $\mathcal{C}^{\mathsf{irr}}_{str}$ for the set of all irreflexive data constructors of str. $\mathcal{C}_{str} := \mathcal{C}^{\mathsf{refl}}_{str} \cup \mathcal{C}^{\mathsf{irr}}_{str}$ denotes the set of all str-constructors.*

Example 2.38. The data structure definitions of Figure 2.1 define the following constructors:

- $\mathsf{0}$ is an irreflexive data constructor with $Occ_{\mathbb{N}}(\mathsf{0}) = \emptyset$.

  $^+(\ldots)$ is a reflexive data constructor with $Occ_{\mathbb{N}}(^+) = \{(1, \epsilon)\}$.

- ε is an irreflexive data constructor with $Occ_{list}(\varepsilon) = \emptyset$.

  :: is a reflexive data constructor, because $Occ_{list}(::) = \{(2, \epsilon)\}$. Furthermore, $Occ_{@A}(::) = \{(1, \epsilon), (2, \mathbf{1})\}$.

- "$\bullet$" is an irreflexive data constructor with $Occ_{pair}(\bullet) = \emptyset$, $Occ_{@A}(\bullet) = \{(1, \epsilon)\}$, and $Occ_{@B}(\bullet) = \{(2, \epsilon)\}$.

- *var* is an irreflexive data constructor with $Occ_{term}(var) = \emptyset$ as well as $Occ_{@V}(var) = \{(1, \epsilon)\}$ and $Occ_{@F}(var) = \emptyset$.

  apply is a reflexive data constructor with $Occ_{term}(apply) = \{(2, \mathbf{1})\}$, $Occ_{@F}(apply) = \{(1, \epsilon), (2, \mathbf{12})\}$, and $Occ_{@V}(apply) = \{(2, \mathbf{11})\}$. $\diamondsuit$

Inferring $\Omega(P)$ from $\Sigma(P)$. The type signature $\Omega(P)$ of a program P can be inferred from the term signature $\Sigma(P)$ of P: Each type constructor *str* occurs in $cons_1 : \tau_{1,1} \times \ldots \times \tau_{1,n_1} \to str[@A_1, \ldots, @A_k]$, because an admissible data structure definition defines at least one data constructor $cons_1$. From this occurrence of *str* one can infer $str \in \Omega(P)_k$.

Therefore, we usually just mention the term signature $\Sigma(P)$, for instance in $\mathcal{T}(\Sigma(P), \mathcal{V})$.

Notation for *if*-, *case*-, and *let*-expressions. In *case*-expressions we separate the first argument t (i. e., the condition) from the m branches $t_1, \ldots, t_m$ by a semicolon. Each branch t_i is written as "$cons_i : t_i$" to indicate the case that t_i belongs to:

$$case\,\{t; cons_1 : t_1, \ldots, cons_m : t_m\}$$

A shorthand notation can be used if some of the branches t_i are equal: Let $\{\{i_1, \ldots, i_k\}, \{i_{k+1}, \ldots, i_m\}\}$ be a partition of $\{1, \ldots, m\}$. Then

$$case\,\{t; cons_{i_1} : t_1, \ldots, cons_{i_k} : t_k, other : t'\}$$

abbreviates

$$case\,\{t; cons_{i_1} : t_1, \ldots, cons_{i_k} : t_k, cons_{i_{k+1}} : t', \ldots, cons_{i_m} : t'\}\,.$$

For case analyses via *if*- or *case*-expressions and for *let*-expressions in procedures, we also use a more verbose notation as it is used in several other programming languages (e. g., ML, Haskell, and Pascal). Table 2.1 lists the notation variants. We mainly employ the functional notation except for the definition of procedures.

The use of curly brackets "{...}" instead of parentheses "(...)" has no special meaning; it merely makes it easier for the human reader to find the matching brackets in such terms.

Table 2.1: Notation of *if*-, *case*-, and *let*-expressions

functional notation	procedural notation
$\mathit{if}\{b, t_1, t_2\}$	*if* b *then* t_1 *else* t_2 *end*
$\mathit{case}\{t; \mathit{cons}_1 : t_1, \ldots, \mathit{cons}_m : t_m\}$	*case* t *of* $\mathit{cons}_1 : t_1,$ $\ldots,$ $\mathit{cons}_m : t_m$ *end*
$\mathit{let}\{x := t;\ r\}$	*let* $x :=$ t *in* r *end*

2.1.4 Procedure Definitions

A procedure definition defines a new function symbol. Additionally, it defines "code" to compute the return value of the procedure. This code is given by a term.

Definition 2.39 (Procedure definitions)**.** *An expression of the form*

$$\texttt{procedure}\ \mathit{proc}(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau <= \texttt{assume}\ c_{\mathit{proc}};\ B_{\mathit{proc}}$$

for $n \geq 1$ and pairwise different identifiers x_i is called a procedure definition *for an $\mathcal{L}$-program P iff*

1. $\mathit{proc} \notin \Sigma(P)_{\tau'}$ *for all* $\tau' \in \mathit{Types}(\Omega(P), \mathcal{W})$,
2. $\tau \in \mathit{Types}(\Omega(P), \mathcal{W})_0$ *such that each type variable in* τ *also appears in at least one of the* τ_i,
3. $\tau_i \in \big(\mathit{Types}(\Omega(P), \mathcal{W})_0 \cup \mathit{Types}(\Omega(P), \mathcal{W})_1\big) \setminus \{\mathit{bool}\}$ *for all* $i = 1, \ldots, n$,
4. $c_{\mathit{proc}} \in \mathcal{T}(\Sigma(P), \{x_1 : \tau_1, \ldots, x_n : \tau_n\})_{\mathit{bool}}$ *is a normalized term,*
5. $B_{\mathit{proc}} \in \mathcal{T}(\Sigma(P'), \{x_1 : \tau_1, \ldots, x_n : \tau_n\})_\tau$ *is a normalized term, where* $\Sigma(P')$ *extends* $\Sigma(P)$ *by* $\mathit{proc} : \tau_1 \times \ldots \times \tau_n \to \tau$,
6. *for each subterm* $\mathit{proc}(t_1, \ldots, t_n)$ *of* B_{proc}, t_i *needs to be of type* τ_i *for all* $i = 1, \ldots, n$.

The x_i are called the formal parameters of *proc. B_{proc} is called the* body of *proc. If $c_{\mathit{proc}} = \mathit{true}$, then "*`assume` *true;" may be omitted in the procedure definition.*

An admissible procedure definition extends the term signature $\Sigma(P)$ to a term signature $\Sigma(P') \supset \Sigma(P)$ by $\mathit{proc} : \tau_1 \times \ldots \times \tau_n \to \tau$.

The requirements of Definition 2.39 mean that

1. symbol *proc* must not have already been used,
2. the return type is a base type (see Section 2.2 for a discussion),
3. *proc* is either a first-order or a second-order procedure,
4. the context requirement of *proc* is not allowed to contain a recursive call of *proc*,
5. the body of *proc* may contain recursive calls, and
6. each recursive call must use arguments of exactly the same type as the formal parameter types; i. e., types may not be specialized in a recursive call.

If function symbol *proc* occurs in body B_{proc} of procedure *proc*, we say that procedure *proc* is defined *recursively*. If it occurs inside a λ-expression that is passed as an argument to a second-order procedure, we say that *proc* is defined by *second-order recursion*. Note that λ-expressions can only occur as arguments of second-order procedures in a procedure body, because B_{proc} is a normalized term and thus β-normal in particular.

Definition 2.40 (Recursion)**.** *A procedure*

$$\texttt{procedure}\ proc(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau <= \texttt{assume}\ c_{proc};\ B_{proc}$$

is defined recursively *iff* $proc \in \Sigma(B_{proc})$*. A subterm* $t = proc(t_1, \ldots, t_n)$ *of* B_{proc} *is a* direct recursive call *iff* t *is a direct function call; otherwise* t *is an* indirect recursive call*. Procedure proc is defined by* second-order recursion *iff* B_{proc} *contains an indirect recursive call.*

Example 2.41. Figure 2.2 shows some procedure definitions. Each $\mathcal{L}$-program P contains by default procedure $>$ that decides if the natural number denoted by x is greater than the natural number denoted by y.[3]

Procedure "<>" concatenates two lists. Procedure "!!" returns the n-th element of list k, where the first element has index `0`. The context requirement $|\,k\,| > n$ demands that n be a valid list index. ♢

Both procedures and λ-expressions have in common that they denote functions for which a "defining term" (called *body*) is specified. We call such functions *body functions*:

Definition 2.42 (Body functions)**.** *For a program* P *we define the family* $\mathcal{T}(\Sigma(P))^{\mathsf{body}} \subset \mathcal{T}(\Sigma(P))$ *by* $t \in \mathcal{T}(\Sigma(P))^{\mathsf{body}}$ *iff* t *is a* λ*-expression or* $t = proc$ *for a procedure* $proc \in \Sigma(P)$*. We say that a term* $t \in \mathcal{T}(\Sigma(P))$ *is a* body function *iff* $t \in \mathcal{T}(\Sigma(P))^{\mathsf{body}}$*.*

[3] We assume procedure $>$ as predefined so that we can use it in termination hypotheses, see Section 4.1.

```
procedure [infix*, 0] >(x, y : ℕ) : ℕ <=
if ?0(x)
  then false
  else if ?0(y)
         then true
         else ⁻(x) > ⁻(y)
       end
end

procedure [infixr, 10] <>(k, l : list[@A]) : list[@A] <=
if ?ε(k)
  then l
  else hd(k) :: (tl(k) <> l)
end

procedure [infix*, 10] !!(k : list[@A], n : ℕ) : @A <=
assume | k | > n;
if ?0(n)
  then hd(k)
  else tl(k) !! ⁻(n)
end
```

Figure 2.2: Procedures ">" (*greater than* on $\mathbb{N}$), "<>" (list concatenation), and "!!" (list access by index)

We write $proc >_{uses} proc'$ iff the context requirement or the body of procedure *proc* contain a call of procedure $proc' \neq proc$:

Definition 2.43 (Relation $>_{uses}$)**.** *Procedure proc* uses *procedure proc′, written* $proc >_{uses} proc'$*, iff* $proc' \in \Sigma(c_{proc}) \cup \Sigma(B_{proc})$ *and* $proc' \neq proc$*.*

Example 2.44. For the procedures in Figure 1.5 (p. 9) we have

- *groundterm* $>_{uses}$ *every* and
- *subterm* $>_{uses}$ *some*.

For procedure "!!" in Figure 2.2 we have "!!" $>_{uses}$ ">", because procedure ">" is used in the context requirement of procedure "!!". ♢

2.1.5 Lemma Definitions

As motivated in Section 1.2, we use universally quantified formulas to specify properties of programs:

Definition 2.45 (Formula)**.** *A* formula *over a term signature* Σ *is an expression of the form* $\forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ b$*, where*

1. $n \in \mathbb{N}$,
2. *the* x_i *are pairwise different identifiers,*
3. $\tau_i \in \big(Types(\Omega(P), \mathcal{W})_0 \cup Types(\Omega(P), \mathcal{W})_1\big) \setminus \{bool\}$ *for all* $i = 1, \ldots, n$*, and*
4. $b \in \mathcal{T}(\Sigma, \{x_1 : \tau_1, \ldots, x_n : \tau_n\})_{bool}$.

Lemmas are "named formulas":

Definition 2.46 (Lemma definitions)**.** *An expression of the form*

`lemma` *name* $<=$ ϕ

is called a lemma definition *for an* $\mathcal{L}$*-program* P *iff* ϕ *is a formula over* $\Sigma(P)$*.*

Example 2.47. We state a lemma that *map* returns a list of the same length as the input list:

`lemma` *map keeps length* $<=$
$\forall f : @A \to @B,\ k : list[@A].\ |\ map(f, k)\ | = |\ k\ |$

The lemma quantifies over variables of a function type and of a base type, respectively. ♢

Table 2.2: Expressing logical connectives with *if*-expressions

formula notation	*if*-notation
$\neg\, a$	$if\{a, false, true\}$
$a \wedge b$	$if\{a, b, false\}$
$a \vee b$	$if\{a, true, b\}$
$a \to b$	$if\{a, b, true\}$
$a \leftrightarrow b$	$if\{a, b, if\{b, false, true\}\}$

When we use the usual logical connectives $\neg$, $\wedge$, $\vee$, $\to$, and $\leftrightarrow$, these abbreviate *if*-expressions as shown in Table 2.2, which are subsequently normalized. For negated equations $\neg\, x \mathbin{=} y$ we write $x \neq y$ as usual.

Example 2.48. Formula

$$\forall t_1, t_2, t_3 : term[@\,V, @F].$$
$$subterm(t_1, t_2) \wedge subterm(t_2, t_3) \to subterm(t_1, t_3)$$

abbreviates

$$\forall t_1, t_2, t_3 : term[@\,V, @F].$$
$$if\{if\{subterm(t_1, t_2), subterm(t_2, t_3), false\}, subterm(t_1, t_3), true\}\,.$$

Normalization of the body of the lemma yields:

$$\forall t_1, t_2, t_3 : term[@\,V, @F].$$
$$if\{subterm(t_1, t_2), if\{subterm(t_2, t_3), subterm(t_1, t_3), true\}, true\} \quad \diamondsuit$$

Boolean terms represent propositions. If a Boolean term does not contain a conditional expression, it is *atomic*:

Definition 2.49 (Atoms, literals, and clauses)**.** *An* atom *or* positive literal *is a term* $b \in \mathcal{T}(\Sigma, \mathcal{V})_{bool} \setminus \{false\}$ *with* $\Sigma_{\mathsf{f}}(b) \cap \Sigma^{\mathsf{cond}} = \emptyset$*. A* negative literal *is either* $b = false$ *or* $b = \neg\, b'$ *for an atom* $b' \neq true$*.* $\mathcal{AT}(\Sigma, \mathcal{V})$ *denotes the set of atoms, and* $\mathcal{LIT}(\Sigma, \mathcal{V})$ *denotes the set of* literals*:*

$$\mathcal{LIT}(\Sigma, \mathcal{V}) := \{\, b \in \mathcal{T}(\Sigma, \mathcal{V})_{bool} \mid b \text{ is a positive or a negative literal} \}$$

A clause *is a finite set* $C \subset_{fin} \mathcal{LIT}(\Sigma, \mathcal{V})$ *of literals. We write* $\mathcal{CL}(\Sigma, \mathcal{V})$ *for the set of clauses over* Σ *and* $\mathcal{V}$*.*

We do not assign a particular meaning to clauses. Sometimes we use a clause C to represent a disjunction (in which case we write $\bigvee C$), whereas it may represent a conjunction in other contexts (denoted by $\bigwedge C$).

2.1.6 Terminology

This section introduces the concepts of *result terms* (e. g., of procedures) and *items* (of data structures).

Result terms. A result term of a term t is a subterm of t that determines a possible result of evaluation of t. For instance, $dbl(m)$ and $^-(n) + m$ are result terms of $if\{?0(n), dbl(m), ^-(n) + m\}$:

Definition 2.50 (Result terms)**.** *Let $t \in \mathcal{T}(\Sigma, \mathcal{V})$ be a normalized let-free*[4] *term. A term position $\pi \in TLPos(t)$ is called a* result term position *iff $\Sigma_{\mathsf{f}}(t|_\pi) \cap \Sigma^{\mathsf{cond}} = \emptyset$ and either*

1. *$\pi = \epsilon$ or*
2. *$\pi = \pi' i$ such that $t|_{\pi'} = f(t_1, \ldots, t_n)$ with $f \in \Sigma^{\mathsf{cond}}$ and $i \geq 2$.*

$ResPos(t) \subseteq TLPos(t)$ denotes the set of all result term positions of t. A subterm $t|_\pi$ of t is called a result term of t *iff $\pi \in ResPos(t)$.*

In other words, a result term is a $\leq_{\mathcal{T}}$-maximal *if*- and *case*-free subterm of t outside an *if*- or *case*-condition or λ-expression.

The *call context* $COND(t, \pi) \in \mathcal{CL}(\Sigma, \mathcal{V})$ consists of the conditions that lead to the subterm at position π in a term t:

Definition 2.51 ($COND(t, \pi)$)**.** *The set $COND(t, \pi) \in \mathcal{CL}(\Sigma, \mathcal{V})$ of* conditions *that lead to position $\pi \in TLPos(t)$ in a let-free (not necessarily normalized) term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is defined as:*

$$COND(t, \epsilon) := \emptyset$$

$$COND(if\{t_1, t_2, t_3\}, i\pi) := \begin{cases} COND(t_1, \pi) & \text{if } i = 1 \\ \{t_1\} \cup COND(t_2, \pi) & \text{if } i = 2 \\ \{\neg\, t_1\} \cup COND(t_3, \pi) & \text{if } i = 3 \end{cases}$$

$$COND(case\{t_1; cons_1 : t_2, \ldots, cons_m : t_{m+1}\}, i\pi) := \begin{cases} COND(t_1, \pi) & \text{if } i = 1 \\ \{?cons_{i-1}(t_1)\} \cup COND(t_i, \pi) & \text{if } i \geq 2 \end{cases}$$

$$COND(f(t_1, \ldots, t_n), i\pi) := COND(t_i, \pi) \quad \text{if } f \notin \Sigma^{\mathsf{cond}}$$

Example 2.52. The result term positions of the body B_{len} of procedure *len* (cf. Figure 1.1 on p. 4) are $ResPos(B_{len}) = \{\mathbf{2}, \mathbf{3}\}$. Thus 0 and $^+(len(tl(k)))$ are the result terms of B_{len}. The call contexts of these result terms are $COND(B_{len}, \mathbf{2}) = \{?\varepsilon(k)\}$ and $COND(B_{len}, \mathbf{3}) = \{\neg\, ?\varepsilon(k)\}$. ♢

[4] For the sake of simplicity, we often assume that terms are *let*-free. A term t can always be made *let*-free by replacing each subterm $let\{x := t_1;\ t_0\}$ of t with $t_0[x/t_1]$. Assuming that a term is *let*-free simplifies matters, because we do not need to keep track of the bindings $\{x/t_1\}$ of *let*-variables.

Example 2.53. The result term positions of the body $B_{subterm}$ of procedure *subterm* (cf. Figure 1.5 on p. 9) are $ResPos(B_{subterm}) = \{\mathbf{2}, \mathbf{32}, \mathbf{33}\}$. Thus *true*, *false*, and $some(\lambda s : term[@V, @F].\, subterm(r, s),\ args(t))$ are the result terms of $B_{subterm}$. The call contexts of these result terms are $COND(B_{subterm}, \mathbf{2}) = \{r = t\}$, $COND(B_{subterm}, \mathbf{32}) = \{r \neq t,\ ?var(t)\}$, and $COND(B_{subterm}, \mathbf{33}) = \{r \neq t,\ ?apply(t)\}$. ♢

When analyzing procedures, we distinguish between the *base cases* and the *recursive cases* of the procedure body. For recursive cases, we are particularly interested in the (positions of the) recursive calls. Since we sometimes need to examine subterms t of the procedure body, we generally define:

Definition 2.54 ($\Pi^{base}_{proc}(t)$ and $\Pi^{rec}_{proc}(t)$)**.** *Let t be a subterm of a let-free procedure body B_{proc}. A term position π of a result term of t denotes a* base case *if neither the result term nor the conditions that lead to this result term contain a recursive call:*

$$\Pi^{base}_{proc}(t) := \{\pi \in ResPos(t) \mid \\ proc \notin \Sigma(t|_\pi) \text{ and } proc \notin \Sigma(COND(t, \pi))\}$$

The term positions of recursive calls *of proc in t are defined by*

$$\Pi^{rec}_{proc}(t) := \{\, \pi \in Pos(t) \mid t|_\pi = proc(\ldots)\} \ .$$

Example 2.55. The positions of the base cases of procedure *subterm* (cf. Figure 1.5 on p. 9) are $\Pi^{base}_{subterm}(B_{subterm}) = \{\mathbf{2}, \mathbf{32}\}$. The positions of the recursive calls of *subterm* are $\Pi^{rec}_{subterm}(B_{subterm}) = \{\mathbf{3310}\}$. This term position denotes the recursive call $subterm(r, s)$. ♢

Items. Lists $k : list[\tau]$ contain items x of type τ. We generalize this terminology and say that some $t : str[\tau_1, \ldots, \tau_k]$ (potentially) contains *items* x_1 of type τ_1, x_2 of type τ_2, etc. Generally, we speak of the items $Itm_\tau(t, \pi)$ of $t : \tau$ at a type position $\pi \in Pos(\tau)$. In order to capture multiple occurrences of items in t, we define $Itm_\tau(t, \pi)$ as a multiset of terms.[5] We write $\{\!|t_1, \ldots, t_n|\!\}$ for the multiset that contains $t_1, \ldots, t_n$ (which need not be pairwise different). Apart from this, we use the same notation for multisets that is customary for sets; e. g., the union of multisets M_1 and M_2 is written as $M_1 \cup M_2$ (and formally means that for each x, one adds the number of occurrences of x in M_1 and in M_2).

[5] A *multiset* is a set that can contain more than one occurrence of an element. Formally, a *set* S of elements from a universe U is a mapping $S : U \to \{0, 1\}$. S contains x, written $x \in S$, iff $S(x) = 1$. A *multiset* M of elements from a universe U is a mapping $M : U \to \mathbb{N}$. $M(x)$ denotes the number of occurrences of $x \in U$ in M.

Definition 2.56 (Items at a type position)**.** *For each ground base type* $\tau = str[\tau'_1, \ldots, \tau'_k]$*, each type position* $\pi \in Pos(\tau)$*, and each term* $t \in \mathcal{T}(\Sigma^{\mathsf{c}})_\tau$ *the multiset* $Itm_\tau(t, \pi) \subseteq \mathcal{T}(\Sigma^{\mathsf{c}})_{\tau|_\pi}$ *of* items of t at type position π *is defined by*

$$Itm_\tau(cons(t_1, \ldots, t_n), \pi) := \begin{cases} \{\!|cons(t_1, \ldots, t_n)|\!\} & \text{if } \pi = \epsilon, \\ \bigcup_{(j,\pi') \in Occ_{@A_h}(cons)} Itm_{\theta(\tau_j)}(t_j, \pi'\pi'') & \text{if } \pi = h\pi'', \end{cases}$$

where $\theta := \{@A_1/\tau'_1, \ldots, @A_k/\tau'_k\}$ *instantiates the type variables of data structure* $str[@A_1, \ldots, @A_k]$*, and str-constructor cons is assumed to be defined by*

$$cons(sel_1 : \tau_1, \ldots, sel_n : \tau_n)\,.$$

If type τ *is clear from the context, we usually omit the type index in* Itm_τ*.*

In case $\pi = h\pi''$ we only collect those items that correspond to the instantiation of $@A_h$ with τ'_h (i. e., $\tau_j\|_{\pi'} = @A_h$), see Example 2.57 below. Intuitively, the multiset $Itm_\tau(t, \pi)$ of items of a term $t \in \mathcal{T}(\Sigma(P)^{\mathsf{c}})_\tau$ at type position π is computed as follows: We replicate the type (and data) constructor definitions so that each type constructor occurs at most once in type τ. Then $Itm_\tau(t, \pi)$ collects the subterms of type $\tau|_\pi$ in t. For instance, $pair[\mathbb{N}, \mathbb{N}]$ is transformed into $pair[\mathbb{N}_1, \mathbb{N}_2]$, so $Itm(t, \mathbf{1})$ collects the subterms of type $\mathbb{N}_1$ in t and $Itm(t, \mathbf{2})$ collects the subterms of type $\mathbb{N}_2$ in t.

Example 2.57. For type $\tau := pair[\mathbb{N}, \mathbb{N}]$, we instantiate type variables $@A$ and $@B$ of data structure $pair[@A, @B]$ by $\theta := \{@A/\mathbb{N}, @B/\mathbb{N}\}$ and get

$$Itm_{pair[\mathbb{N},\mathbb{N}]}((\mathsf{x} \bullet \mathsf{y}), \mathbf{1}) = Itm_{\mathbb{N}}(\mathsf{x}, \epsilon) = \{\!|\mathsf{x}|\!\}$$

as well as

$$Itm_{pair[\mathbb{N},\mathbb{N}]}((\mathsf{x} \bullet \mathsf{y}), \mathbf{2}) = Itm_{\mathbb{N}}(\mathsf{y}, \epsilon) = \{\!|\mathsf{y}|\!\}\ . \qquad \diamondsuit$$

Example 2.58. For type $\tau := list[\mathbb{N}]$, we instantiate type variable $@A$ of data structure $list[@A]$ by $\theta := \{@A/\mathbb{N}\}$ and get

$$\begin{aligned} & Itm_{list[\mathbb{N}]}(4 :: 1 :: 4 :: \varepsilon, \mathbf{1}) \\ = {} & Itm_{\theta(@A)}(4, \epsilon) \cup Itm_{\theta(list[@A])}(1 :: 4 :: \varepsilon, \mathbf{1}) \\ = {} & \{\!|4|\!\} \cup Itm_{\theta(@A)}(1, \epsilon) \cup Itm_{\theta(list[@A])}(4 :: \varepsilon, \mathbf{1}) \\ = {} & \{\!|4|\!\} \cup \{\!|1|\!\} \cup Itm_{\theta(@A)}(4, \epsilon) \cup Itm_{\theta(list[@A])}(\varepsilon, \mathbf{1}) \\ = {} & \{\!|4, 1|\!\} \cup \{\!|4|\!\} \cup \emptyset = \{\!|4, 1, 4|\!\}\ . \end{aligned}$$

Thus $Itm_{list[\mathbb{N}]}(\mathsf{k}, \mathbf{1})$ collects the items of list k, whereas

$$Itm_{list[\mathbb{N}]}(4 :: 1 :: 4 :: \varepsilon, \epsilon) = \{\!|4 :: 1 :: 4 :: \varepsilon|\!\}$$

returns the whole list. $\diamondsuit$

For a type position $\pi = \pi_1\pi_2$, $Itm(t, \pi)$ can be computed from $Itm(t, \pi_1)$ as follows:

Lemma 2.59. *For all ground base types τ, all type positions $\pi_1, \pi_2 \in \mathbb{N}^*$ with $\pi_1\pi_2 \in Pos(\tau)$, and all terms $t \in \mathcal{T}(\Sigma^{\mathsf{c}})_\tau$:*

$$Itm(t, \pi_1\pi_2) = \bigcup_{t' \in Itm(t,\pi_1)} Itm(t', \pi_2)$$

Proof. If $\pi_1 = \epsilon$, then $Itm(t, \pi_1) = \{\!|t|\!\}$, so the equality trivially holds.

If $\pi_1 \neq \epsilon$, we prove the statement by structural induction on t.

$t = cons$: $Itm(t, \pi_1\pi_2) = \emptyset$ and $Itm(t, \pi_1) = \emptyset$.

$t = cons(t_1, \ldots, t_n)$: Let $\pi_1 = h\pi''$ for some $h \in \mathbb{N}$ and $\pi'' \in \mathbb{N}^*$. The induction hypothesis is

$$\text{(IH)} \quad Itm(t_i, \pi_1'\pi_2') = \textstyle\bigcup_{t' \in Itm(t_i,\pi_1')} Itm(t', \pi_2')$$

for all $i \in \{1, \ldots, n\}$, π_1', and π_2'. Hence:

$$\begin{aligned}
&\textstyle\bigcup_{t' \in Itm(t,h\pi'')} Itm(t', \pi_2) \\
&= \textstyle\bigcup_{(j,\pi') \in Occ_{@A_h}(cons)} \bigcup_{t' \in Itm(t_j,\pi'\pi'')} Itm(t', \pi_2) && \text{; by def. of } Itm \\
&= \textstyle\bigcup_{(j,\pi') \in Occ_{@A_h}(cons)} Itm(t_j, \pi'\pi''\pi_2) && \text{; by (IH)} \\
&= Itm(t, h\pi''\pi_2) && \text{; by def. of } Itm
\end{aligned}$$

as desired. □

Example 2.60. For type $\tau := list[pair[\mathbb{N}, \mathbb{N}]]$, we can collect the items of the first pair components and of the second pair components using Lemma 2.59:

$$\begin{aligned}
&Itm_{list[pair[\mathbb{N},\mathbb{N}]]}\big((3 \bullet 4) :: (5 \bullet 6) :: \varepsilon, \mathbf{11}\big) \\
&= Itm_{pair[\mathbb{N},\mathbb{N}]}\big((3 \bullet 4), \mathbf{1}\big) \cup Itm_{pair[\mathbb{N},\mathbb{N}]}\big((5 \bullet 6), \mathbf{1}\big) \\
&= \{\!|3|\!\} \cup \{\!|5|\!\} = \{\!|3, 5|\!\}
\end{aligned}$$

$$\begin{aligned}
&Itm_{list[pair[\mathbb{N},\mathbb{N}]]}\big((3 \bullet 4) :: (5 \bullet 6) :: \varepsilon, \mathbf{12}\big) \\
&= Itm_{pair[\mathbb{N},\mathbb{N}]}\big((3 \bullet 4), \mathbf{2}\big) \cup Itm_{pair[\mathbb{N},\mathbb{N}]}\big((5 \bullet 6), \mathbf{2}\big) \\
&= \{\!|4|\!\} \cup \{\!|6|\!\} = \{\!|4, 6|\!\}
\end{aligned}$$

♢

One easily observes that all items of a term t are subterms of t:

Lemma 2.61. *For all ground base types τ, terms $t \in \mathcal{T}(\Sigma^{\mathsf{c}})$, and type positions $\pi \in Pos(\tau)$, $t' \leq_{\mathcal{T}} t$ for all $t' \in Itm_\tau(t, \pi)$. If $\pi \neq \epsilon$, then $t' <_{\mathcal{T}} t$ for all $t' \in Itm_\tau(t, \pi)$.*

Proof. The claim follows from the definition of $Itm_\tau(t, \pi)$ by structural induction on t. □

```
procedure apply.renaming(t : term[@V, @F],
                         σ : @V → @V) : term[@V, @F] <=
case t of
  var   : var(σ(vsym(t))),
  apply : apply(fsym(t),
                map(λs : term[@V, @F]. apply.renaming(s, σ), args(t)))
end
```

Figure 2.3: Second-order recursion in a second-order procedure

2.2 Expressive Power of $\mathcal{L}$

In Chapter 1 we already presented several examples that characterize the expressive power of $\mathcal{L}$. Therefore this section is mainly devoted to examples that *cannot* be expressed in $\mathcal{L}$.

Before getting to these "negative examples", let us investigate the expressive power of second-order recursion. Procedure *groundterm* in Figure 1.5 (p. 9) is a *first-order* procedure that is defined by second-order recursion. Second-order recursion is a feature that is *not* limited to first-order procedures:

Defining second-order procedures by second-order recursion. In $\mathcal{L}$ it is possible to define *second-order* procedures by second-order recursion: In Figure 2.3, procedure *apply.renaming* gets a function σ as parameter which maps variable symbols to variable symbols. Procedure *apply.renaming* applies σ to all variable symbols in term t. It passes the λ-expression

$$\lambda s : term[@V, @F].\ apply.renaming(s, \sigma)$$

of type $term[@V, @F] \to term[@V, @F]$ to the second-order procedure *map*.

This definition is possible, because procedure *apply.renaming* essentially operates on term $t : term[@V, @F]$. Type $term[@V, @F]$ is a base type, because we demand that type variables be instantiated with base types only.

This brings us to some examples that *cannot* be expressed in $\mathcal{L}$.

No instantiation of type variables with function types. Procedure *map* (Figure 1.3 on p. 6) can easily be considered as a higher-order procedure by allowing that $@A$ may be instantiated with a type of arbitrary order. This would facilitate the uniform transformation of functions. For instance, $map\big(\lambda g : \mathbb{N} \to \mathbb{N}.\ (\lambda k : list[@A].\ g(|\,k\,|)),\ dbl :: half :: \varepsilon\big)$ returns a list of the following two functions:

- $\lambda k : list[@A].\ dbl(|\,k\,|)$
- $\lambda k : list[@A].\ half(|\,k\,|)$

```
procedure [infix*, 5] ∈ (x : @A, k : list[@A]) : bool <=
if ?ε(k)
  then false
  else if hd(k) = x
         then true
         else x ∈ tl(k)
       end
end
```

Figure 2.4: Procedure "$\in$" to decide if a list contains a certain element

If we allowed the instantiation of type variables with function types, we would run into the following problem: It is in general undecidable if two functions f and g are equal. However, we expect that procedure "$\in$" in Figure 2.4) denotes a computable function. But $hd(k) = x$ would be undecidable if $@A$ was allowed to be a function type, so procedure "$\in$" would not denote an algorithm that can be run on a computer.

In Haskell [54], instantiation of type variables with function types is basically allowed. Whenever equality of functions needs to be decided, a runtime error occurs. For instance, it is impossible to evaluate $dbl \in dbl :: \varepsilon$.

In order to avoid reasoning about such runtime errors, we choose to restrict the language so that each terminating procedure denotes a total function.

No functions as return values. Since the result type of a procedure needs to be a base type (cf. Definition 2.39 on p. 36), one cannot define procedures such as

```
procedure compose(f : @B → @C, g : @A → @B) : @A → @C <=
λx : @A. f(g(x))
```

and

```
procedure neg(p : @A → bool) : @A → bool <=
λx : @A. ¬ p(x)
```

in $\mathcal{L}$.[6] These procedures are not defined recursively. In theorem proving, such procedures are difficult to handle—not only for an automated theorem prover, but also for a human. The reason is that it requires a certain expertise to decide when it is best to replace such a procedure call with the instantiated body. Since one can always write $\lambda x : @A. \neg p(x)$ instead of

[6] As another example, a curried implementation $+ : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ of addition cannot be defined in $\mathcal{L}$. (Definition 2.1 on p. 18 would consider $+$ as a second-order procedure in contrast to the first-order implementation in Figure 1.6 on p. 11.)

```
procedure funpow(n : ℕ, f : @A → @A, x : @A) : @A <=
if ?0(n)
  then x
  else funpow(⁻(n), f, f(x))
end
```

Figure 2.5: Procedure *funpow* iterates a function a fixed number of times

$neg(p)$, this restriction is only a matter of convenience in this case, not of expressivity.

Of course, there are also procedures that are defined recursively and return a function. For instance, the following procedure *funpow′* returns $f^n = \underbrace{f \circ \ldots \circ f}_{n \text{ times}}$:

```
procedure funpow′(n : ℕ, f : @A → @A) : @A → @A <=
if ?0(n)
  then λx : @A. x
  else λx : @A. f(funpow′(⁻(n), f)(x))
end
```

However, this is a rather awkward definition. Usually, one defines *funpow* as in Figure 2.5 and writes

$$\lambda x : @A.\, funpow(n, f, x)$$

whenever the iterated function itself is needed.

The restriction becomes more pronounced when we consider the return types of *selectors*. For example, the following data structure cannot be defined in $\mathcal{L}$, because selector *wt.children* does not return a base type:

```
structure wide.tree[@A] <=
  wt.tip,
  wt.node(wt.value : @A, wt.children : ℕ → wide.tree[@A])
```

This data structure would represent infinitely branching trees. In order to prove statements about such infinite objects, *co-induction* would be an appropriate technique [48], whereas we consider theorem proving by *induction*.

2.3 Semantics

During the execution of a program, the variables of the programs (e. g., the formal parameters) are assigned a *value*. The values of a ground type τ are defined as follows:

Definition 2.62 (Values)**.** *For an $\mathcal{L}$-program P and a ground type $\tau \in Types(\Omega(P))$, $\mathbb{V}(P)_\tau$ denotes the* values of type τ*:*

- *For each ground base type $\tau = str[\tau_1, \ldots, \tau_k]$, $\mathbb{V}(P)_\tau$ contains all constructor ground terms of type τ:*

$$\mathbb{V}(P)_\tau := \mathcal{T}(\Sigma(P)^{\mathsf{c}})_\tau$$

- *For each ground function type $\tau = \tau_1 \times \ldots \times \tau_k \to \tau_{k+1}$, $\mathbb{V}(P)_\tau$ contains all closed λ-expressions of type τ:*

$$\mathbb{V}(P)_\tau := \{\, \lambda x_1 : \tau_1, \ldots, x_k : \tau_k.\, t \mid\ t \in \mathcal{T}(\Sigma(P), \{x_1, \ldots, x_k\})\}$$

Example 2.63. Let P be an $\mathcal{L}$-program that contains the definitions given in Figures 1.1, 1.3, 1.5, and 1.6. We list some examples of values:

- $\mathbb{V}(P)_{bool} = \{true, false\}$
- $\mathbb{V}(P)_{\mathbb{N}} \supsetneq \{\mathsf{0}, {}^{+}(\mathsf{0}), {}^{+}({}^{+}(\mathsf{0}))\} = \{0, 1, 2\}$
- $\mathbb{V}(P)_{list[\mathbb{N}]} \supsetneq \{\varepsilon,\ 0 :: \varepsilon,\ 1 :: \varepsilon,\ 2 :: \varepsilon,\ 0 :: 0 :: \varepsilon,\ 0 :: 1 :: \varepsilon\}$
- $\mathbb{V}(P)_{pair[\mathbb{N},\mathbb{N}]} \supsetneq \{(0 \bullet 0),\ (0 \bullet 1),\ (1 \bullet 0),\ (1 \bullet 1),\ (0 \bullet 2)\}$
- $\mathbb{V}(P)_{\mathbb{N}\to\mathbb{N}} \supsetneq \{\lambda n : \mathbb{N}.\, \mathsf{0},\ \lambda n : \mathbb{N}.\, {}^{+}(n),\ \lambda x : \mathbb{N}.\, x + 3\}$
- $\mathbb{V}(P)_{term[\mathbb{N},\mathbb{N}]\to term[\mathbb{N},\mathbb{N}]} \supsetneq \{\lambda t : term[\mathbb{N},\mathbb{N}].\, t,\ \lambda t : term[\mathbb{N},\mathbb{N}].\, var(1)\}$
- $\mathbb{V}(P)_{term[\mathbb{N},\mathbb{N}]\to bool} \supsetneq$
  $\{\lambda t : term[\mathbb{N},\mathbb{N}].\, groundterm(t),\ \lambda t : term[\mathbb{N},\mathbb{N}].\, subterm(var(3), t)\}$ $\diamondsuit$

For each ground type τ there exists at least one value, so $\mathbb{V}(P)_\tau \neq \emptyset$. We extend the definition of so-called *witness values* from [88] to ground function types as follows:

Definition 2.64 (Witness values)**.** *For an $\mathcal{L}$-program P and a ground type $\tau \in Types(\Omega(P))$, $\omega_\tau \in \mathbb{V}(P)_\tau$ denotes the* witness value of type τ*:*

- *For type $\tau = bool$, $\omega_{bool} := false$.*
- *For a ground base type $\tau = str[\tau_1, \ldots, \tau_k]$ and a data structure definition as in Definition 2.31 (p. 30),*

$$\omega_\tau := cons_i(\omega_{\theta(\tau_{i,1})}, \ldots, \omega_{\theta(\tau_{i,n_i})})\,,$$

  where $\theta := \{@A_1/\tau_1, \ldots, @A_k/\tau_k\}$ and $i \in \{1, \ldots, m\}$ is the smallest number such that $n_i = 0$ or—if no such i exists—the smallest number such that $cons_i$ is irreflexive.

- *For a ground function type $\tau = \tau_1 \times \ldots \times \tau_k \to \tau_{k+1}$,*

$$\omega_\tau := \lambda x_1 : \tau_1, \ldots, x_k : \tau_k.\, \omega_{\tau_{k+1}}\,.$$

Example 2.65. Let P be an $\mathcal{L}$-program that contains the data structure definitions of Figure 2.1 (p. 31). We list some examples of witness values:

$$\begin{array}{ll}
\omega_{\mathbb{N}} = \mathsf{0} & \omega_{pair[\mathbb{N},term[\mathbb{N},\mathbb{N}]]} = (\mathsf{0} \bullet var(\mathsf{0})) \\
\omega_{list[\mathbb{N}]} = \varepsilon & \omega_{\mathbb{N}\to\mathbb{N}} = \lambda x\!:\!\mathbb{N}.\,\mathsf{0} \\
\omega_{pair[\mathbb{N},\mathbb{N}]} = (\mathsf{0} \bullet \mathsf{0}) & \omega_{term[\mathbb{N},\mathbb{N}]\to bool} = \lambda x\!:\!term[\mathbb{N},\mathbb{N}].\,false \\
\omega_{term[\mathbb{N},\mathbb{N}]} = var(\mathsf{0}) & \omega_{term[\mathbb{N},\mathbb{N}]\to term[\mathbb{N},\mathbb{N}]} = \lambda x\!:\!term[\mathbb{N},\mathbb{N}].\,var(\mathsf{0}) \quad \diamondsuit
\end{array}$$

Notation of values. We often denote an arbitrary value by q. For instance, if $x_1, \ldots, x_n$ are the parameters of a procedure that are instantiated with values $q_1, \ldots, q_n$, we write $\{x_1/q_1, \ldots, x_n/q_n\}$ for this instantiation. Sometimes, however, it is more convenient to use variable symbols like f for a variable of some function type and p for a variable that denotes a predicate (i. e., p has function type "$\ldots \to bool$"). In this case we denote the value that such a variable is assigned with by f, p, and x; e. g., $\{f/\mathsf{f}, p/\mathsf{p}, x/\mathsf{x}\}$.

2.3.1 Computation Calculus

The operational semantics of $\mathcal{L}$ is defined by an *interpreter*

$$eval_P : \mathcal{T}(\Sigma(P)) \mapsto \mathbb{V}(P)\,.$$

The interpreter tries to evaluate a ground term $t \in \mathcal{T}(\Sigma(P))_\tau$ to a value $eval_P(t) \in \mathbb{V}(P)_\tau$ of the same type as t. The evaluation of t may diverge if a procedure that is called in t does not terminate. Hence $eval_P$ is a *partial* mapping and we write $eval_P(t) = \bot$ if evaluation of t diverges. If $eval_P(t) \in \mathbb{V}(P)$, we say that evaluation of t *terminates*.

The computation steps of the interpreter $eval_P$ are defined by the so-called *computation calculus*:

Definition 2.66 (Computation calculus)**.** *For an $\mathcal{L}$-program P, the* computation calculus *is defined by:*

Language: $\mathcal{T}(\Sigma(P))$

Inference Rules: *The inference rules of the computation calculus (called* computation rules*) are shown in Figure 2.6. Each computation rule is of the form*

$$\frac{t}{t'} \;, \textit{if } \mathsf{cond}[t]$$

such that $t' \in \mathcal{T}(\Sigma(P))_\tau$ whenever $t \in \mathcal{T}(\Sigma(P))_\tau$ for some ground type τ. A computation rule may only be applied if side condition $\mathsf{cond}[t]$ *is satisfied.*

Deduction: *We write $t \Rightarrow_P t'$ iff t' results from t by applying some computation rule. $\Rightarrow_P^*$ denotes the reflexive and transitive closure of $\Rightarrow_P$. We write $t \Rightarrow_P^! t'$ iff $t \Rightarrow_P^* t'$ and $t' \not\Rightarrow_P t''$ for all $t'' \in \mathcal{T}(\Sigma(P))$.*

The computation rules in Figure 2.6 merge and extend the corresponding definitions from [72, 88, 90, 97]:

- Function calls $f(t_1, \ldots, t_n) : \tau$ that violate the context requirement of f are evaluated to the witness value ω_τ of type τ, cf. computation rule (4), which is taken over from [88]. Since procedures in [88] have no context requirements, we have modified computation rule (15) correspondingly to consider the case that the context requirement of a procedure may be violated.

  This simplifies the definition of the semantics from [72, 90, 97], where the result of a function call is an arbitrary (but undefined) value if the context requirement is violated. Since we statically ensure via proof obligations (so-called *context hypotheses*, see Section 3.5) that the context requirement of each function call in a program is satisfied, we can simply use a well-defined witness value as result.

- In contrast to [72, 88, 90, 97], the computation calculus of Definition 2.66 is deterministic: To each term $t \in \mathcal{T}(\Sigma(P))$ at most one computation rule is applicable. The side condition of computation rule (14) enforces that the arguments of function calls (where f may also be a λ-expression) are evaluated from left to right.[7]

- As in [90], the side conditions of the computation rules demand that the arguments be evaluated before evaluating the call of the leading function (symbol), so we get *call-by-value* evaluation.

- Computation rule (16) is new and implements β-reduction, cf. Definition 2.21 (p. 27).

If t is a constructor ground term or a λ-expression, no computation rule is applicable to t: $t \not\Rightarrow_P t'$ for all $t \in \mathbb{V}(P)$ and $t' \in \mathcal{T}(\Sigma(P))$. This means that values are not evaluated further.[8] Moreover, the following lemma shows that any terminating evaluation yields a value.

[7] Without commitment to the left-to-right evaluation order, one can show *confluence* of $\Rightarrow_P$ [88]. Thus the evaluation order does not matter so we may simply fix a particular evaluation order.

[8] Although a λ-expression t is not evaluated further by the computation calculus in Figure 2.6, it is harmless to "evaluate" it to any other λ-expression t' with $eval_P(t(q_1, \ldots, q_n)) = eval_P(t'(q_1, \ldots, q_n))$ for all $q_1, \ldots, q_n \in \mathbb{V}(P)$. In practice, it is beneficial to normalize λ-expressions at least. See Section 5.4 for details.

$$(1)\ \frac{?cons_i(cons_i(q_1,\ldots,q_{n_i}))}{true}\ \text{, if } q_1,\ldots,q_{n_i}\in\mathbb{V}(P)$$

$$(2)\ \frac{?cons_{i'}(cons_i(q_1,\ldots,q_{n_i}))}{false}\ \text{, if } q_1,\ldots,q_{n_i}\in\mathbb{V}(P) \text{ and } cons_{i'}\neq cons_i$$

$$(3)\ \frac{sel_{i,j}(cons_i(q_1,\ldots,q_{n_i}))}{q_j}\ \text{, if } q_1,\ldots,q_{n_i}\in\mathbb{V}(P)$$

$$(4)\ \frac{sel_{i',j}(cons_i(q_1,\ldots,q_{n_i}))}{\omega_\tau}\ \text{, if } q_1,\ldots,q_{n_i}\in\mathbb{V}(P) \text{ and } i'\neq i$$

$$(5)\ \frac{q_1 = q_2}{true}\ \text{, if } q_1,q_2\in\mathbb{V}(P) \text{ and } q_1=q_2$$

$$(6)\ \frac{q_1 = q_2}{false}\ \text{, if } q_1,q_2\in\mathbb{V}(P) \text{ and } q_1\neq q_2$$

$$(7)\ \frac{if\{b,t_1,t_2\}}{if\{b',t_1,t_2\}}\ \text{, if } b\Rightarrow_P b'$$

$$(8)\ \frac{if\{true,t_1,t_2\}}{t_1}$$

$$(9)\ \frac{if\{false,t_1,t_2\}}{t_2}$$

$$(10)\ \frac{case\,\{t;cons_1:t_1,\ldots,cons_m:t_m\}}{case\,\{t';cons_1:t_1,\ldots,cons_m:t_m\}}\ \text{, if } t\Rightarrow_P t'$$

$$(11)\ \frac{case\,\{cons_i(q_1,\ldots,q_{n_i});cons_1:t_1,\ldots,cons_m:t_m\}}{t_i}\ \text{, if } q_1,\ldots,q_{n_i}\in\mathbb{V}(P)$$

$$(12)\ \frac{let\{x := t;\ r\}}{let\{x := t';\ r\}}\ \text{, if } t\Rightarrow_P t'$$

$$(13)\ \frac{let\{x := q;\ r\}}{r[x/q]}\ \text{, if } q\in\mathbb{V}(P)$$

$$(14)\ \frac{f(t_1,\ldots,t_{i-1},t_i,t_{i+1},\ldots,t_n)}{f(t_1,\ldots,t_{i-1},t_i',t_{i+1},\ldots,t_n)}\ \text{, if } f\notin\Sigma(P)^{\mathsf{cond}},\ t_i\Rightarrow_P t_i',\ \text{and } t_1,\ldots,t_{i-1}\in\mathbb{V}(P)$$

$$(15)\ \frac{proc(q_1,\ldots,q_n)}{if\{c_{proc},B_{proc},\omega_\tau\}[x_1/q_1,\ldots,x_n/q_n]}\ \text{, if } q_1,\ldots,q_n\in\mathbb{V}(P)$$

$$(16)\ \frac{(\lambda x_1:\tau_1,\ldots,x_n:\tau_n.t)(q_1,\ldots,q_n)}{t[x_1/q_1,\ldots,x_n/q_n]}\ \text{, if } q_1,\ldots,q_n\in\mathbb{V}(P)$$

Figure 2.6: Inference rules of the computation calculus

Lemma 2.67. *Whenever $t \Rightarrow_P^! t'$ for some ground terms $t, t' \in \mathcal{T}(\Sigma(P))_\tau$, then $t' \in \mathbb{V}(P)_\tau$.*

Proof. One easily proves by structural induction on t' that for each $t' \in \mathcal{T}(\Sigma(P)) \setminus \mathbb{V}(P)$ there is some term t'' with $t' \Rightarrow_P t''$. Thus a term $t' \in \mathcal{T}(\Sigma(P)) \setminus \mathbb{V}(P)$ is never minimal wrt. $\Rightarrow_P$. □

Definition 2.68 (Interpreter)**.** *For an $\mathcal{L}$-program P, the* interpreter $eval_P : \mathcal{T}(\Sigma(P)) \mapsto \mathbb{V}(P)$ *is defined by*

$$eval_P(t) := \begin{cases} t' & \text{if } t \Rightarrow_P^! t' \text{ for some } t' \in \mathbb{V}(P) \\ \bot & \text{if } t \not\Rightarrow_P^! t' \text{ for all } t' \in \mathbb{V}(P) \ . \end{cases}$$

Example 2.69. For procedure *dbl* (Figure 1.6 on p. 11) we compute the following evaluation. We indicate the subterm that changes by underlining.

$$\begin{aligned}
& \underline{dbl(^+(^+(0)))} \\
\Rightarrow_P\ & \underline{if\{true, if\{?0(^+(^+(0))), 0, ^+(^+(dbl(^-(^+(^+(0))))))\}, \omega_{\mathbb{N}}\}} && \text{; by (15)} \\
\Rightarrow_P\ & if\{\underline{?0(^+(^+(0)))}, 0, ^+(^+(dbl(^-(^+(^+(0))))))\} && \text{; by (8)} \\
\Rightarrow_P\ & \underline{if\{false, 0, ^+(^+(dbl(^-(^+(^+(0))))))\}} && \text{; by (7)+(2)} \\
\Rightarrow_P\ & ^+(^+(dbl(\underline{^-(^+(^+(0)))})))) && \text{; by (9)} \\
\Rightarrow_P\ & ^+(^+(\underline{dbl(^+(0))})) && \text{; by (14)+(3)} \\
\Rightarrow_P\ & ^+(^+(\underline{if\{true, if\{?0(^+(0)), 0, ^+(^+(dbl(^-(^+(0)))))\}, \omega_{\mathbb{N}}\}})) && \text{; by (14)+(15)} \\
\Rightarrow_P\ & ^+(^+(if\{\underline{?0(^+(0))}, 0, ^+(^+(dbl(^-(^+(0)))))\})) && \text{; by (14)+(8)} \\
\Rightarrow_P\ & ^+(^+(\underline{if\{false, 0, ^+(^+(dbl(^-(^+(0)))))\}})) && \text{; by (14)+(7)+(2)} \\
\Rightarrow_P\ & ^+(^+(^+(^+(dbl(\underline{^-(^+(0))})))))) && \text{; by (14)+(9)} \\
\Rightarrow_P\ & ^+(^+(^+(^+(\underline{dbl(0)})))) && \text{; by (14)+(3)} \\
\Rightarrow_P\ & ^+(^+(^+(^+(\underline{if\{true, if\{?0(0), 0, ^+(^+(dbl(^-(0))))\}, \omega_{\mathbb{N}}\}})))) && \text{; by (14)+(15)} \\
\Rightarrow_P\ & ^+(^+(^+(^+(if\{\underline{?0(0)}, 0, ^+(^+(dbl(^-(0))))\})))) && \text{; by (14)+(8)} \\
\Rightarrow_P\ & ^+(^+(^+(^+(\underline{if\{true, 0, ^+(^+(dbl(^-(0))))\}})))) && \text{; by (14)+(7)+(1)} \\
\Rightarrow_P\ & ^+(^+(^+(^+(0)))) && \text{; by (14)+(8)}
\end{aligned}$$

Thus $eval_P(dbl(^+(^+(0)))) = ^+(^+(^+(^+(0))))$, i. e., $eval_P(dbl(2)) = 4$. ♢

Example 2.70. For procedure *groundterm* (Figure 1.5 on p. 9) and arbitrary values $\mathsf{f}, \mathsf{g}, \mathsf{v} \in \mathbb{V}(P)_{\mathbb{N}}$ we compute the following evaluations. We write groundterm for the λ-expression $\lambda s : term[\mathbb{N}, \mathbb{N}].\ groundterm(s)$.

$$\begin{aligned}
& groundterm(var(\mathsf{v})) \\
\Rightarrow_P\ & if\{true, case\{var(\mathsf{v});\ var : false,\ apply : every(\ldots)\}, \omega_{bool}\} \\
\Rightarrow_P\ & case\{var(\mathsf{v});\ var : false,\ apply : every(\ldots)\} \\
\Rightarrow_P\ & false
\end{aligned}$$

We abbreviate the following evaluations by using $\Rightarrow_P^*$, which leaves out some trivial computations such as $\mathit{if}\{?\varepsilon(\ldots :: \ldots), t_1, t_2\} \Rightarrow_P^* t_2$.

$$
\begin{aligned}
& \mathit{groundterm}(\mathit{apply}(\mathsf{g}, \varepsilon)) \\
\Rightarrow_P^* \; & \mathit{case}\{\mathit{apply}(\ldots);\ \ldots,\ \mathit{apply} : \mathit{every}(\mathsf{groundterm}, \mathit{args}(\mathit{apply}(\mathsf{g}, \varepsilon)))\} \\
\Rightarrow_P \; & \mathit{every}(\mathsf{groundterm}, \mathit{args}(\mathit{apply}(\mathsf{g}, \varepsilon)) \\
\Rightarrow_P \; & \mathit{every}(\mathsf{groundterm}, \varepsilon) \\
\Rightarrow_P^* \; & \mathit{if}\{?\varepsilon(\varepsilon), \mathit{true}, \ldots\} \\
\Rightarrow_P \; & \mathit{if}\{\mathit{true}, \mathit{true}, \ldots\} \\
\Rightarrow_P \; & \mathit{true}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{groundterm}(\mathit{apply}(\mathsf{f}, \mathit{apply}(\mathsf{g}, \varepsilon) :: \mathit{var}(\mathsf{v}) :: \varepsilon)) \\
\Rightarrow_P^* \; & \mathit{case}\{\mathit{apply}(\ldots);\ \ldots,\ \ \mathit{apply} : \mathit{every}(\mathsf{groundterm}, \mathit{args}(\ldots))\} \\
\Rightarrow_P \; & \mathit{every}(\mathsf{groundterm}, \mathit{args}(\mathit{apply}(\mathsf{f}, \mathit{apply}(\mathsf{g}, \varepsilon) :: \mathit{var}(\mathsf{v}) :: \varepsilon))) \\
\Rightarrow_P \; & \mathit{every}(\mathsf{groundterm}, \mathit{apply}(\mathsf{g}, \varepsilon) :: \mathit{var}(\mathsf{v}) :: \varepsilon) \\
\Rightarrow_P^* \; & \mathit{if}\{(\lambda s : \mathit{term}[\mathbb{N}, \mathbb{N}].\ \mathit{groundterm}(s))(\mathit{apply}(\mathsf{g}, \varepsilon)), \ldots, \ldots\} \\
\Rightarrow_P \; & \mathit{if}\{\mathit{groundterm}(\mathit{apply}(\mathsf{g}, \varepsilon)), \ldots, \ldots\} \\
\Rightarrow_P^* \; & \mathit{if}\{\mathit{true}, \mathit{every}(\mathsf{groundterm}, \mathit{tl}(\mathit{apply}(\mathsf{g}, \varepsilon) :: \mathit{var}(\mathsf{v}) :: \varepsilon)), \ldots\} \\
\Rightarrow_P \; & \mathit{every}(\mathsf{groundterm}, \mathit{tl}(\mathit{apply}(\mathsf{g}, \varepsilon) :: \mathit{var}(\mathsf{v}) :: \varepsilon)) \\
\Rightarrow_P \; & \mathit{every}(\mathsf{groundterm}, \mathit{var}(\mathsf{v}) :: \varepsilon) \\
\Rightarrow_P^* \; & \mathit{if}\{\mathit{groundterm}(\mathit{var}(\mathsf{v})), \mathit{every}(\ldots), \mathit{false}\} \\
\Rightarrow_P^* \; & \mathit{if}\{\mathit{false}, \mathit{every}(\ldots), \mathit{false}\} \\
\Rightarrow_P \; & \mathit{false}
\end{aligned}
$$

Thus $\mathit{eval}_P\big(\mathit{groundterm}(\mathit{apply}(\mathsf{f}, \mathit{apply}(\mathsf{g}, \varepsilon) :: \mathit{var}(\mathsf{v}) :: \varepsilon))\big) = \mathit{false}$, which makes sense, because "$\mathsf{f}(\mathsf{g}, \mathsf{v})$" is not a ground term. ♢

We write $t_1 \approx t_2$ iff terms t_1 and t_2 are semantically equivalent:

Definition 2.71 (Semantic equivalence $\approx$)**.** *Let P be an $\mathcal{L}$-program and let $\mathcal{V} = \{x_1, \ldots, x_n\}$ be a finite family of variables with $x_i : \tau_i$ for $i = 1, \ldots, n$. Terms $t_1, t_2 \in \mathcal{T}(\Sigma(P), \mathcal{V})$ are* semantically equivalent*, written $t_1 \approx t_2$, iff*

$$\mathit{eval}_P(t_1[q_1, \ldots, q_n]) = \mathit{eval}_P(t_2[q_1, \ldots, q_n])$$

holds for all grounding type substitutions $\theta \in \mathit{GndSubst}_{\Omega(P)}(\tau_1, \ldots, \tau_n)$ and all values $q_1, \ldots, q_n$ with $q_i \in \mathbb{V}(P)_{\theta(\tau_i)}$ for $i = 1, \ldots, n$.

Example 2.72. $\mathit{dbl}(2) \approx 4$ and $\mathit{dbl}(^+(n)) \approx {}^+({}^+(\mathit{dbl}(n)))$ for $n : \mathbb{N}$. ♢

2.3.2 Required Evaluation of Subterms

In Example 2.69, evaluation of the procedure call $dbl(2)$ requires the evaluation of the recursive call $dbl(1)$, which in turn requires evaluation of $dbl(0)$. We write $dbl(2) \rhd dbl(1) \rhd dbl(0)$, where "$\rhd$" is a relation on function calls and means "requires evaluation of". Evaluation of $dbl(1)$ is required to evaluate $dbl(2)$, because the body of procedure dbl contains a recursive call $dbl(^-(n))$ under call context $\{\neg\, ?0(n)\}$. For $n := 2$, the condition in this call context evaluates to *true*.

Before defining relation $\rhd$ formally, let us convince ourselves that the call context $COND(t, \pi)$ as defined in Definition 2.51 (p. 41) indeed contains those conditions that need to be satisfied to evaluate subterm $t|_\pi$ according to the computation rules of Figure 2.6:

For an *if*-expression $t := if\{t_1, t_2, t_3\}$, evaluation of t_1 is always required due to rule (7), so $COND(t, \mathbf{1}) = \emptyset$. Evaluation of t_2 is required iff b evaluates to *true*, cf. rule (8), so $COND(t, \mathbf{2}) = \{t_1\}$. Evaluation of t_3 is required iff b evaluates to *false*, cf. rule (9), so $COND(t, \mathbf{3}) = \{\neg\, t_1\}$.

For a *case*-expression $t := case\{t'; cons_1 : t_1, \ldots, cons_m : t_m\}$, evaluation of t' is always required due to rule (10). Evaluation of a *case*-branch t_i in t is required iff $?cons_i(t')$ evaluates to *true*, cf. rule (11). Again, this corresponds exactly to the definition of $COND(t, \pi)$.

For a function call $t := f(t_1, \ldots, t_n)$ with $f \notin \Sigma(P)^{\mathsf{cond}}$, evaluation of t_i is required iff evaluation of $t_1, \ldots, t_{i-1}$ terminates, cf. rule (14). The definition of $COND(t, \pi)$ ignores this arbitrary choice of left-to-right evaluation and considers evaluation of *all* t_i as required.

Summing up, if the interpreter $eval_P$ evaluates a subterm $t|_\pi$ of a ground term t, then $eval_P(c) = true$ for all $c \in COND(t, \pi)$. Thus $COND(t, \pi)$ is a set of *necessary* conditions for the fact that evaluation of $t|_\pi$ is required. If all sub-evaluations terminate, then $COND(t, \pi)$ is also a set of *sufficient* conditions for the fact that evaluation of $t|_\pi$ is required.

Hence we define the subterms that need to be evaluated as follows:

Definition 2.73 (Required evaluation)**.** *The set $EvalPos_P(t) \subseteq TLPos(t)$ of positions of subterms that need to be evaluated in order to evaluate a let-free ground term $t \in \mathcal{T}(\Sigma(P))$ is defined by*

$$EvalPos_P(t) := \{\pi \in TLPos(t) \mid eval_P(c) = true \text{ for all } c \in COND(t, \pi)\} \ .$$

Evaluation of term $t' \in \mathcal{T}(\Sigma(P))$ is required *in order to evaluate a let-free ground term $t \in \mathcal{T}(\Sigma(P))$, written $t \rightsquigarrow_P t'$, iff there exists a term position $\pi \in EvalPos_P(t)$ with $t|_\pi = t'$.*

Based on the required evaluation positions, we now define the call relation $\rhd$. The intuitive meaning of $f(q_1, \ldots, q_n) \rhd g(q'_1, \ldots, q'_m)$ is that the

evaluation of function call $f(q_1, \ldots, q_n)$ requires the evaluation of function call $g(q'_1, \ldots, q'_m)$ in the instantiated body or context requirement of f.

Definition 2.74 (Call relation $\rhd$)**.** *The* call relation $\rhd$ *on* $\mathcal{T}(\Sigma(P))$ *is defined by* $t \rhd t'$ *iff*

1. $t = f(q_1, \ldots, q_n) \in \mathcal{T}(\Sigma(P))_\tau$ *for values* $q_i \in \mathbb{V}(P)$ *and a body function* $f \in \mathcal{T}(\Sigma(P))^{\mathsf{body}}$ *with normalized let-free body* B_f*, context requirement*[9] c_f*, and parameters* $x_1, \ldots, x_n$*,*
2. $t' = g(q'_1, \ldots, q'_m) \in \mathcal{T}(\Sigma(P))_{\tau'}$ *for* $m \geq 1$ *values* $q'_j \in \mathbb{V}(P)$ *and a term* $g \in \mathcal{T}(\Sigma(P))$*, and*
3. *there exist terms* $h, t'_1, \ldots, t'_m$ *such that*
   - $\mathit{if}\{c_f, B_f, \omega_\tau\} \rightsquigarrow_P h(t'_1, \ldots, t'_m)$*,*
   - $h[x_1/q_1, \ldots, x_n/q_n] =_\eta g$*, and*
   - $\mathit{eval}_P(t'_j[x_1/q_1, \ldots, x_n/q_n]) = q'_j$ *for all* $j = 1, \ldots, m$*.*

Example 2.75. We have $\mathit{dbl}(2) \rhd \mathit{dbl}(1)$, because $\mathit{eval}_P(\neg ?\mathsf{0}(n)[n/2]) = \mathit{true}$, $\mathit{dbl}[n/2] =_\eta \mathit{dbl}$, and $\mathit{eval}_P({}^-(n)[n/2]) = 1$.

For procedure *groundterm* (cf. Figure 1.5 on p. 9) and a term $\mathsf{t} := \mathit{apply}(\mathsf{f}, \mathit{apply}(\mathsf{g}, \varepsilon) :: \mathit{var}(\mathsf{v}) :: \varepsilon) \in \mathbb{V}(P)_{\mathit{term}[\mathbb{N},\mathbb{N}]}$ for $\mathsf{f}, \mathsf{g}, \mathsf{v} \in \mathbb{V}(P)_{\mathbb{N}}$, we get

$$\begin{aligned}
&\mathit{groundterm}(\mathit{apply}(\mathsf{f}, \mathit{apply}(\mathsf{g}, \varepsilon) :: \mathit{var}(\mathsf{v}) :: \varepsilon)) \\
&\quad \rhd \mathit{every}(\lambda s : \mathit{term}[\mathbb{N}, \mathbb{N}].\, \mathit{groundterm}(s),\ \mathit{apply}(\mathsf{g}, \varepsilon) :: \mathit{var}(\mathsf{v}) :: \varepsilon) \\
&\quad \rhd \mathit{groundterm}(\mathit{apply}(\mathsf{g}, \varepsilon)) \\
&\quad \rhd \mathit{every}(\lambda s : \mathit{term}[\mathbb{N}, \mathbb{N}].\, \mathit{groundterm}(s),\ \varepsilon)
\end{aligned}$$

and also

$$\begin{aligned}
&\mathit{groundterm}(\mathit{apply}(\mathsf{f}, \mathit{apply}(\mathsf{g}, \varepsilon) :: \mathit{var}(\mathsf{v}) :: \varepsilon)) \\
&\quad \rhd \mathit{every}(\lambda s : \mathit{term}[\mathbb{N}, \mathbb{N}].\, \mathit{groundterm}(s),\ \mathit{apply}(\mathsf{g}, \varepsilon) :: \mathit{var}(\mathsf{v}) :: \varepsilon) \\
&\quad \rhd \mathit{every}(\lambda s : \mathit{term}[\mathbb{N}, \mathbb{N}].\, \mathit{groundterm}(s),\ \mathit{var}(\mathsf{v}) :: \varepsilon) \\
&\quad \rhd \mathit{groundterm}(\mathit{var}(\mathsf{v}))\,.
\end{aligned}$$

If we replaced $=_\eta$ in Definition 2.74 with $=$, we would get some additional intermediate steps due to η-expanded arguments: For instance,

$$\begin{aligned}
&\mathit{every}(\lambda s : \mathit{term}[\mathbb{N}, \mathbb{N}].\, \mathit{groundterm}(s),\ \mathit{var}(\mathsf{v}) :: \varepsilon) \\
&\quad \rhd \big(\lambda s : \mathit{term}[\mathbb{N}, \mathbb{N}].\, \mathit{groundterm}(s)\big)(\mathit{var}(\mathsf{v})) \\
&\quad \rhd \mathit{groundterm}(\mathit{var}(\mathsf{v}))\,.
\end{aligned}$$

We prefer the shorter $\rhd$-sequence above (i. e., the definition with $=_\eta$), as we are mainly interested in procedure calls, not in calls of λ-expressions. ♢

[9] The context requirement of a λ-expression f is $c_f := \mathit{true}$.

The call relation $\rhd$ allows us to find out which function calls $g(q'_1, \ldots, q'_m)$ are required to evaluate a procedure call $f(q_1, \ldots, q_n)$. If we are interested in the calls of a particular function g, we write $\rhd_g$:

Definition 2.76 (Call relation $\rhd_g$)**.** *For a term $g \in \mathcal{T}(\Sigma(P))_\tau \setminus \Sigma(P)^{\mathsf{cond}}$ of a function type τ, the* call relation $\rhd_g$ *on $\mathcal{T}(\Sigma(P))$ is defined by $t \rhd_g t'$ iff*

1. *$t' = g(q'_1, \ldots, q'_m)$ for $m \geq 1$ values $q'_j \in \mathbb{V}(P)$ and*
2. *$t \rhd h_1(\ldots) \rhd \ldots \rhd h_k(\ldots) \rhd t'$ for some $h_i \in \mathcal{T}(\Sigma(P))$ such that $h_i \neq_\eta g$ for all $i = 1, \ldots, k$, where $k \geq 0$.*

We write $Calls_g(t)$ for the set of all function calls $g(\ldots)$ that need be evaluated to evaluate a ground term $t \in \mathcal{T}(\Sigma(P))$:

$$\begin{aligned} &Calls_g(t) := \\ &\qquad \big\{\, g(q'_1, \ldots, q'_m) \;\big|\; t \rightsquigarrow_P g(q'_1, \ldots, q'_m) \text{ for some } q'_j \in \mathbb{V}(P) \big\} \cup \\ &\qquad \big\{\, t'' \in \mathcal{T}(\Sigma(P)) \;\big|\; t \rightsquigarrow_P t' \rhd_g t'' \text{ for some } t' \in \mathcal{T}(\Sigma(P)) \big\} \end{aligned}$$

Thus $t \rhd_g t'$ iff t' is the first call of g in a sequence of nested function calls.

Example 2.77. Following up Example 2.75, we have $dbl(2) \rhd_{dbl} dbl(1) \rhd_{dbl} dbl(0)$ as well as

$$\begin{aligned} &groundterm(\mathsf{t}) \rhd_{groundterm} groundterm(apply(\mathsf{g}, \varepsilon))\,, \\ &groundterm(\mathsf{t}) \rhd_{groundterm} groundterm(var(\mathsf{v}))\,, \\ &groundterm(\mathsf{t}) \rhd_{every} \\ &\quad every(\lambda s : term[\mathbb{N}, \mathbb{N}].\, groundterm(s),\ apply(\mathsf{g}, \varepsilon) :: var(\mathsf{v}) :: \varepsilon)\,, \text{ and} \\ &groundterm(\mathsf{t}) \not\rhd_{every} \\ &\quad every(\lambda s : term[\mathbb{N}, \mathbb{N}].\, groundterm(s),\ var(\mathsf{v}) :: \varepsilon) \end{aligned}$$

for $\mathsf{t} := apply(\mathsf{f}, apply(\mathsf{g}, \varepsilon) :: var(\mathsf{v}) :: \varepsilon)$. ♢

For the termination analysis of some procedure f, the *recursive calls* $f(q'_1, \ldots, q'_n)$ are particularly important:

Definition 2.78 (Recursive call relation)**.** *For a procedure $f : \tau_1 \times \ldots \times \tau_n \to \tau$ of a program P and a type substitution $\theta \in GndSubst_{\Omega(P)}(\tau_1, \ldots, \tau_n)$, the* recursive call relation $\succ_f^\theta$ *on $\mathbb{V}(P)_{\theta(\tau_1)} \times \ldots \times \mathbb{V}(P)_{\theta(\tau_n)}$ is defined by $(q_1, \ldots, q_n) \succ_f^\theta (q'_1, \ldots, q'_n)$ iff $f(q_1, \ldots, q_n) \rhd_f f(q'_1, \ldots, q'_n)$.*

Example 2.79. For procedure dbl, $3 \succ_{dbl} 2 \succ_{dbl} 1 \succ_{dbl} 0 \not\succ_{dbl} n$ for all $n \in \mathbb{V}(P)_{\mathbb{N}}$.

For procedure *groundterm*, term t from Example 2.75, and type substitution $\theta := \{@V/\mathbb{N}, @F/\mathbb{N}\}$, the recursive calls are $\mathsf{t} \succ^\theta_{groundterm} apply(\mathsf{g}, \varepsilon)$ and $\mathsf{t} \succ^\theta_{groundterm} var(\mathsf{v})$. ♢

2.3.3 Termination

Intuitively, a procedure f terminates iff evaluation of $f(q_1, \ldots, q_n)$ terminates for all $q_i \in \mathbb{V}(P)$. This definition of termination works well for first-order procedures f. If f is a second-order procedure, however, at least one of the q_i is a λ-expression. The body of this λ-expression may contain procedure calls $g(\ldots)$. Even if f is a "terminating" procedure, evaluation of $f(q_1, \ldots, q_n)$ may diverge solely because g does not terminate. Consequently, we should only demand that the evaluation of $f(q_1, \ldots, q_n)$ terminate if g terminates.

Formally, we let $P_\downarrow$ denote the subprogram of P with only terminating procedures. We restrict the q_i to be elements of $\mathbb{V}((P \setminus \{f\})_\downarrow)$, so λ-expressions may only contain calls $g(\ldots)$ of *terminating* procedures $g \neq f$. In short, evaluation of $f(q_1, \ldots, q_n)$ needs to terminate for all "terminating inputs" q_i.

Given a terminating second-order procedure f of a program P, we expect that evaluation of $f(q_1, \ldots, q_n)$ also terminates if a λ-expression q_i contains a call of a terminating procedure g that is not defined in P, but in an extension P' of P. In other words, we expect *monotonicity* in the sense that procedure f remains terminating when new procedures are added to program P. Hence the following definition of termination also considers such extensions $P' \supseteq P$ of the program P that f is defined in.[10]

Definition 2.80 (Termination)**.** *Let P be an $\mathcal{L}$-program that contains a procedure definition* `procedure` $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau$ `<= assume` c_f; B_f.

Procedure f of program P terminates*, written $f \in P_\downarrow$, iff for all programs $P' \supseteq P$, all type substitutions $\theta \in GndSubst_{\Omega(P')}(\tau_1, \ldots, \tau_n)$, and all values $q_1, \ldots, q_n$ with $q_i \in \mathbb{V}((P' \setminus \{f\})_\downarrow)_{\theta(\tau_i)}$ for all $i = 1, \ldots, n$,*

$$eval_{P'}(f(q_1, \ldots, q_n)) \in \mathbb{V}(P')_{\theta(\tau)} .$$

Program P terminates *iff all procedures f defined in P terminate (i. e., if $P_\downarrow = P$).*

By Definition 2.80, evaluation of $f(q_1, \ldots, q_n)$ terminates for all $q_i \in \mathbb{V}((P' \setminus \{f\})_\downarrow)$ that do *not* contain calls of f, if f terminates. Of course, then evaluation of $f(q_1, \ldots, q_n)$ also terminates if f *is* called in some λ-expression: Define an extension $P' \supseteq P$ that contains a copy f' of procedure f. Replace all occurrences of f in the q_i with f'. Let q_i' be the results of this replacement. By Definition 2.80, evaluation of $f(q_1', \ldots, q_n')$ terminates. Since f' computes the same function as f, evaluation of $f(q_1, \ldots, q_n)$ terminates as well.

Usually we show termination of a procedure f by assuming an arbitrary $\mathcal{L}$-program P that contains procedure f and by checking the requirements

[10] Of course, program P' may also define additional data structures that can be used when instantiating the type variables in the signature of f.

of Definition 2.80 for program P (e. g., we use $eval_P$ instead of $eval_{P'}$). This simplifies the notation and is obviously equivalent to assuming an arbitrary extension P' of a fixed program P and checking the requirements for program P'.

The evaluation of a term terminates if and only if computation rule (15) is applied only finitely many times during the evaluation. If all procedures $g \neq f$ in a program P are known to terminate, then termination of f solely depends on whether the evaluation of an arbitrary f-call $f(q_1, \ldots, q_n)$ requires the evaluation of only finitely many recursive f-calls. Formally, this means that relation $\succ_f$ is *well-founded* (see [88], for instance):

Definition 2.81 (Well-founded relations)**.** *A relation $\succ \subseteq S \times S$ is* well-founded *iff there is no infinite sequence $(s_i)_{i\in\mathbb{N}}$ of elements $s_i \in S$ such that $s_i \succ s_{i+1}$ for all $i \in \mathbb{N}$.*

Example 2.82. The usual *greater than* relation $> \subset \mathbb{N}\times\mathbb{N}$ is well-founded. Relation $\geq \subset \mathbb{N} \times \mathbb{N}$ is *not* well-founded, because $0 \geq 0 \geq 0 \ldots$ ♢

Lemma 2.83. *Let P be an arbitrary $\mathcal{L}$-program that contains a procedure $f : \tau_1 \times \ldots \times \tau_n \to \tau$. If all procedures $g \in P$ with $f >^+_{uses} g$ terminate,*[11] *then procedure f terminates iff $\succ^\theta_f$ is a well-founded relation for each type substitution $\theta \in GndSubst_{\Omega(P)}(\tau_1, \ldots, \tau_n)$.*

Proof. We prove the equivalence by considering the two implications separately:

"⇒": Procedure f terminates, so $eval_P(f(q_1, \ldots, q_n)) \neq \bot$ for all values $q_i \in \mathbb{V}(P)_{\theta(\tau_i)}$ for $i = 1, \ldots, n$. Each sequence

$$f(q_1, \ldots, q_n) \rhd t' \rhd t'' \rhd \ldots$$

is finite, because each $\rhd$-step corresponds to one $\Rightarrow_P$-step (namely an application of computation rule (15)), and the $\Rightarrow_P$-deduction is finite due to $eval_P(f(q_1, \ldots, q_n)) \neq \bot$. Consequently, each sequence

$$f(q_1, \ldots, q_n) \rhd_f f(q'_1, \ldots, q'_n) \rhd_f f(q''_1, \ldots, q''_n) \rhd_f \ldots$$

needs to be finite by definition of $\rhd_f$. Thus $\succ^\theta_f$ is well-founded.

"⇐": Relation $\succ^\theta_f$ is well-founded, so each sequence

$$f(q_1, \ldots, q_n) \rhd_f f(q'_1, \ldots, q'_n) \rhd_f f(q''_1, \ldots, q''_n) \rhd_f \ldots$$

is finite. Since each $\rhd_f$-step abbreviates a finite $\rhd$-sequence by definition of $\rhd_f$, each sequence

$$f(q_1, \ldots, q_n) \rhd t' \rhd t'' \rhd \ldots$$

[11] $>^+_{uses}$ denotes the transitive closure of $>_{uses}$.

is finite. Relation $\rhd$ is locally finite, i. e., for any t' there are only finitely many t'' (up to $=_\eta$) with $t' \rhd t''$. Evaluation of each $\rhd_f$-minimal function call terminates, because all procedures $g \in P$ with $f >^+_{uses} g$ terminate by assumption. Thus evaluation of $f(q_1, \ldots, q_n)$ requires only finitely many applications of computation rule (15), so $eval_P(f(q_1, \ldots, q_n)) \neq \bot$. □

2.3.4 Truth of Formulas

A formula $\forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ b$ is true iff all instances of b evaluate to *true*:

Definition 2.84 (Truth)**.** *A formula $\forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ b$ over a term signature $\Sigma(P)$ for a terminating program P is* true *iff*

$$eval_{P'}(b[\vec{q}]) = true$$

for all terminating programs $P' \supseteq P$, all grounding type substitutions $\theta \in GndSubst_{\Omega(P')}(\tau_1, \ldots, \tau_n)$, and all values $q_1, \ldots, q_n$ with $q_i \in \mathbb{V}(P')_{\theta(\tau_i)}$ for all $i = 1, \ldots, n$.

A lemma `lemma` *name* $<= \phi$ *is* true *iff formula ϕ is true.*

2.4 Relativized Procedures

According to computation rule (15) of the computation calculus (cf. Figure 2.6 on p. 51), the semantics of a procedure call $proc(q_1, \ldots, q_n)$ is given by $if\{c_{proc}, B_{proc}, \omega_\tau\}$. For some definitions, it is beneficial to assume that a procedure *proc* is defined by

$$\texttt{procedure}\ proc(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau <= B^{\mathsf{rel}}_{proc}\,,$$

where $B^{\mathsf{rel}}_{proc} := Normalize(if\{c_{proc}, B_{proc}, \omega\})$ for a fresh constant $\omega : \tau$ with $eval_P(\omega) := \omega_\tau$. In other words, the context requirement is incorporated into the *relativized procedure body* B^{rel}_{proc} [72] so that we do not need to consider c_{proc} explicitly. Instead, the context requirement is considered implicitly, because it becomes part of the call contexts of subterms of B_{proc}.

Example 2.85. The relativized version of procedure "!!" (cf. Figure 2.2 on p. 38) is given by:

```
procedure [infix*, 10] !!(k : list[@A], n : ℕ) : @A <=
if | k | > n
  then if ?0(n)
         then hd(k)
         else tl(k) !! ⁻(n)
       end
  else ω
end
```

Thus the call context of subterm $hd(k)$ is $\{|\, k \,| > n,\ ?0(n)\}$, for instance. ♢

Incompletely defined procedures. By providing a context requirement c_{proc} for a procedure

$$\texttt{procedure}\ proc(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau <= \texttt{assume}\ c_{proc};\ B_{proc}$$

body B_{proc} only needs to consider the cases where c_{proc} is satisfied. Sometimes such a context requirement can only be formulated using auxiliary procedures.

Example 2.86. Consider a procedure

$$\texttt{procedure}\ retrieve(key : @A,\ assoc : list[pair[@A, @B]]) : @B$$

that is supposed to return item b of the first pair $(a \bullet b)$ in list *assoc* with $a = key$. Clearly, a meaningful result can only be defined if there exists such a pair $(key \bullet b)$ in k. Thus the user would have to implement a procedure

$$\texttt{procedure}\ contains(key : @A,\ assoc : list[pair[@A, @B]]) : bool$$

that can be used as context requirement for procedure *retrieve*:

$$c_{retrieve} := contains(key, assoc) \qquad \diamondsuit$$

Walther and Schweitzer [97] present an approach that facilitates the implementation of *incompletely defined procedures* without requiring the user to implement auxiliary procedures for the context requirement. To this effect, the user may use symbol "$\star$" as result term for those cases that cannot be assigned a meaningful result. Then a so-called *domain procedure* ∇_{proc} is automatically synthesized that is subsequently used as context requirement for *proc*.

Example 2.87. Using the approach from [97], procedure *retrieve* may be *incompletely defined* as in Figure 2.7. Since we cannot define a meaningful result if list *assoc* is empty, we just write "$\star$" as result term for this case.

From this incomplete definition of *retrieve*, domain procedure $\nabla_{retrieve}$ is synthesized, which returns *true* iff no such $\star$-case is reached during evaluation of $retrieve(key, assoc)$. This domain procedure is subsequently used as context requirement for *retrieve*, resulting in procedure *retrieve*$'$ given in Figure 2.7, where the $\star$-cases have been eliminated. $\diamondsuit$

Without the concepts of *incompletely defined procedures* and *context requirements of procedures*, procedure *retrieve* could not be implemented, because one cannot specify a result term of type $@B$ for the case $?\varepsilon(assoc)$.

However, if all types in the signature of a procedure *proc* are *monomorphic*, then we could dispense with these concepts and directly use the relativized body B^{rel}_{proc}.

Example 2.88. The relativized monomorphic instance $\{@A/\mathbb{N}\}$ of procedure "!!" (cf. Figure 2.2 on p. 38 and Example 2.85 above) is given by:

```
procedure [infix*, 10] !!(k : list[ℕ], n : ℕ) : ℕ <=
if | k | > n
  then if ?0(n)
         then hd(k)
         else tl(k) !! ⁻(n)
       end
  else 0
end
```

This implementation is inferior to the original implementation:

1. It just handles monomorphic lists of natural numbers, whereas the original implementation handles polymorphic lists.

2. The "context requirement" $| k | > n$ needs to be checked at run-time in each recursive call. This is inefficient.

3. If $| k | \not> n$, then a quite arbitrary result (namely 0) is returned. Such arbitrary results are often counterintuitive [94]. ♢

To avoid these problems, our approach supports procedures with context requirements. This allows for polymorphic implementations as well as static checks of the context requirements by the verifier, so counterintuitive return values do not occur.

```
procedure retrieve(key : @A, assoc : list[pair[@A, @B]]) : @B <=
if ?ε(assoc)
  then ⋆
  else if key = fst(hd(assoc))
         then snd(hd(assoc))
         else retrieve(key, tl(assoc))
       end
end

procedure ∇retrieve(key : @A, assoc : list[pair[@A, @B]]) : bool <=
if ?ε(assoc)
  then false
  else if key = fst(hd(assoc))
         then true
         else ∇retrieve(key, tl(assoc))
       end
end

procedure retrieve′(key : @A, assoc : list[pair[@A, @B]]) : @B <=
assume ∇retrieve(key, assoc);
if key = fst(hd(assoc))
  then snd(hd(assoc))
  else retrieve′(key, tl(assoc))
end
```

Figure 2.7: Procedure *retrieve* to find an entry in an association list

Chapter 3

Finite Quantification

From a theoretical perspective, quantification over finite domains is "boring", because it is trivially decidable if a formula $\forall x \in M.\ p(x)$ holds if $M = \{m_1, \ldots, m_n\}$ is a finite set and if predicate p is decidable; one simply evaluates the finite conjunction $p(m_1) \wedge \ldots \wedge p(m_n)$.

However, decidability of finite quantifications is precisely what makes them interesting for automated theorem proving in practice. Whenever finite quantification suffices, a formula should be phrased in a way that makes the finite quantification explicit so that it can be exploited.

In this chapter, we describe the uniform synthesis of *quantification procedures* that implement such decision procedures. We focus on *universal* quantification and consider existential quantification ($\exists x \in M.\ p(x)$) only as a notational abbreviation for negated universal quantification ($\neg\forall x \in M.\ \neg p(x)$).

Finite quantification is practically relevant to the synthesis of so-called *context hypotheses* (Section 3.5), to termination analysis of procedures (Section 4.1), and to the synthesis of induction axioms (Sections 5.2 and 5.3).

Organization of this chapter. Section 3.1 presents quantification procedures for polymorphic data structures. Quantification procedures for second-order procedures are defined in Section 3.2. Section 3.3 introduces existential quantification as a notational abbreviation for negated universal quantification. Section 3.4 summarizes the applications of quantification procedures along with some notation. We describe the synthesis of context hypotheses in Section 3.5. Finally, in Section 3.6 we show that a limited number of quantification procedures suffices in practice.

3.1 Quantification Procedures for Data Structures

In mathematics, the notation $\forall x \in M.\ p(x)$ for some (not necessarily finite) set $M \subseteq \mathbb{N}$ is an abbreviation for

$$\forall x.\ x \in M \rightarrow p(x)\ , \tag{3.1}$$

where x is declared somewhere else to be "of type $\mathbb{N}$".[1] In our definition of formulas (Definition 2.45 on p. 39) we included the type declaration for term variables x in the quantification as it is customary in formal logics:

$$\forall x : \mathbb{N}.\ x \in M \rightarrow p(x) \tag{3.2}$$

If set M is finite, deciding if (3.2) holds amounts to iterating over the *members* x of set M and checking $p(x)$ each time.

In Definition 2.56 (p. 42), we generalized the notion of *membership* to so-called *items* of arbitrary data structures. Given a list $k = x_1 :: \ldots :: x_n :: \varepsilon$ of type $\mathit{list}[\tau]$, one might want to know whether $p(x)$ holds for all items x in k:

$$\forall x : \tau.\ x \in_{\mathit{list}} k \rightarrow p(x) \tag{3.3}$$

Here symbol $\in_{\mathit{list}} : @A \times \mathit{list}[@A] \rightarrow \mathit{bool}$ denotes a procedure that decides if some $x : @A$ is an item of a list $k : \mathit{list}[@A]$.

In order to decide if (3.3) is true, we just need to iterate over the (finitely many) items x of k and check if $p(x)$ evaluates to *true*. But how do we know that this intuitive approach ("iterating over the items") suffices? And how is a theorem prover supposed to know that it should "iterate over the items" and prove $p(x)$ for each such item?

Considering just the structure of formula (3.3), a theorem prover would usually try to show $p(x)$ for an arbitrary $x : \tau$ with the hypothesis that $x \in_{\mathit{list}} k$ is true. This is quite different, because it separates checking $p(x)$ from the iteration over the items of k.

The following formula is equivalent to (3.3), but clearly exhibits the iteration over list items:

$$\mathit{every}(p, k) \tag{3.4}$$

Procedure *every* has been introduced in Figure 1.3 (p. 6) as a user-defined procedure. In Section 3.1 we describe the uniform synthesis of quantification procedures *forall.str* for data structures $\mathit{str}[@A_1, \ldots, @A_k]$. For instance, for $\mathit{list}[@A]$ a quantification procedure *forall.list* is synthesized that is equivalent to procedure *every*. In principle, synthesizing *forall.str* is just as simple (or difficult) as synthesizing a generalized membership procedure $\in_{\mathit{str}}$. We discuss the latter alternative in Chapter 7.

[1] Of course, we could have taken any other set S instead of $\mathbb{N}$ in the example.

3.1.1 Uniform Synthesis of *forall.str*

Let P be a program that contains a data structure definition (cf. Definition 2.31 on p. 30) of the form

$$\begin{array}{l}\texttt{structure}\ str[@A_1,\ldots,@A_k]\ <= \\ \quad \ldots, \\ \quad cons(sel_1:\tau_1,\ldots,sel_n:\tau_n), \\ \quad \ldots\end{array} \tag{3.5}$$

Given some $x : str[@A_1,\ldots,@A_k]$ and a predicate $p : @A_h \to bool$ for some $h \in \{1,\ldots,k\}$, the call $forall.str_h(p,x)$ of the quantification procedure for *str* yields *true* iff $p(z)$ holds for all items $z : @A_h$ in x (cf. Lemma 3.5 below).

Definition 3.1 (Quantification procedures for data structures)**.** *For each data structure definition of the form (3.5) that defines a type constructor str with arity $k \geq 1$, the* quantification procedures *$forall.str_h$* for *str and $h = 1,\ldots,k$ are defined by*[2]

$$\begin{array}{l}\texttt{procedure}\ forall.str_h(p : @A_h \to bool, \\ \qquad\qquad\qquad\qquad x : str[@A_1,\ldots,@A_k]) : bool\ <= \\ case\ x\ of \\ \quad \ldots, \\ \quad cons\ :\ \bigwedge_{(j,\pi)\in Occ_{@A_h}(cons)} \texttt{ALL}_{\tau_j}(sel_j(x),\pi), \\ \quad \ldots \\ end\end{array}$$

where

$$\begin{array}{l}\texttt{ALL}_{str'[\tau'_1,\ldots,\tau'_{k'}]}(t,\epsilon) := p(t) \\ \texttt{ALL}_{str'[\tau'_1,\ldots,\tau'_{k'}]}(t,i\pi') := forall.str'_i(\lambda y : \tau'_i.\ \texttt{ALL}_{\tau'_i}(y,\pi'),\ t)\ .\end{array}$$

If type constructor str is unary, we usually omit the index h and just write forall.str.

For some $x : str[@A_1,\ldots,@A_k]$ with $?cons(x)$ for a *str*-constructor *cons*, procedure $forall.str_h$ checks if predicate $p : @A_h \to bool$ is satisfied for all items $z : @A_h$ in $x = cons(sel_1(x),\ldots,sel_n(x))$. Such items z can only occur in some $sel_j(x) : \tau_j$ that contains type $@A_h$ at some position π. $\texttt{ALL}_{\tau_j}(t,\pi)$ represents a case analysis over this position π:[3] If $\pi = \epsilon$, we directly apply p to t, because t is of type $@A_h$. Otherwise $@A_h$ occurs nested at position $\pi = i\pi'$ in τ. In this case we use the quantification procedure $forall.str'$ of the leading type constructor str' of τ_j to iterate through t until we finally reach $@A_h$, i. e., $\pi = \epsilon$.

[2] The finite conjunction can be expressed by *if*-expressions as shown in Table 2.2 (p. 40).

[3] The invariant of $\texttt{ALL}_\tau(t,\pi)$ is: t is of type τ and $\tau|_\pi = @A_h$.

```
procedure forall.list(p : @A → bool, k : list[@A]) : bool <=
case k of
  ε : true,
  :: : if p(hd(k))
         then forall.list(p, tl(k))
         else false
       end
end

procedure forall.pair_1(p : @A → bool, x : pair[@A, @B]) : bool <=
p(fst(x))

procedure forall.pair_2(p : @B → bool, x : pair[@A, @B]) : bool <=
p(snd(x))
```

Figure 3.1: Automatically synthesized quantification procedures for type constructors *list* and *pair*, cf. Figure 2.1 (p. 31)

In the following examples we instantiate Definition 3.1 for data structures $list[@A]$, $pair[@A, @B]$, and $term[@V, @F]$ (cf. Figure 2.1 on p. 31). The resulting quantification procedures are shown in Figures 3.1 and 3.2.

Example 3.2. For $list[@A]$, constructor ε has no selectors, so *forall.list* returns *true* in this case. Constructor :: may contain items $z : @A$ both in $hd(k) : @A$ and in $tl(k) : list[@A]$, so *forall.list* returns the conjunction of

$$\mathtt{ALL}_{@A}(hd(k), \epsilon) = p(hd(k))$$

and

$$\begin{aligned}\mathtt{ALL}_{list[@A]}(tl(k), \mathbf{1}) &= forall.list(\lambda y : @A.\, \mathtt{ALL}_{@A}(y, \epsilon),\ tl(k)) \\ &= forall.list(\lambda y : @A.\, p(y),\ tl(k))\end{aligned}$$

in this case. Procedure *forall.list* is an example of a quantification procedure with a direct recursive call. ♢

Example 3.3. Since there is only one data constructor for $pair[@A, @B]$, there is no need for a case analysis via *case*. Instead, $forall.pair_1$ just returns

$$\mathtt{ALL}_{@A}(fst(x), \epsilon) = p(fst(x))$$

and $forall.pair_2$ returns

$$\mathtt{ALL}_{@B}(snd(x), \epsilon) = p(snd(x)) \ ,$$

because both $@A$ and $@B$ occur exactly once in the selector types of constructor $\bullet$ at type position ϵ. Procedures $forall.pair_1$ and $forall.pair_2$ are examples of quantification procedures that are *not* defined recursively. ♢

```
procedure forall.term1(p : @V → bool, t : term[@V, @F]) : bool <=
case t of
  var   : p(vsym(t)),
  apply : forall.list(λs : term[@V, @F]. forall.term1(p, s), args(t))
end

procedure forall.term2(p : @F → bool, t : term[@V, @F]) : bool <=
case t of
  var   : true,
  apply : if p(fsym(t))
            then forall.list(λs : term[@V, @F]. forall.term2(p, s), args(t))
            else false
          end
end
```

Figure 3.2: Automatically synthesized quantification procedures for type constructor *term*, cf. Figure 2.1 (p. 31)

Example 3.4. For $term[@V, @F]$, case $?var(t)$ is as straightforward as in the previous examples. In case $?apply(t)$, both $forall.term_1$ and $forall.term_2$ call quantification procedure *forall.list* to iterate through the list of direct subterms of t.

For the occurrence of $@V$ in the selector types of constructor *apply*, $forall.term_1$ returns

$$\begin{aligned}
&\mathtt{ALL}_{list[term[@V,@F]],\mathbf{11}}(p, args(t)) \\
&= forall.list(\lambda s : term[@V, @F].\, \mathtt{ALL}_{term[@V,@F]}(s, \mathbf{1}),\ args(t)) \\
&= forall.list(\lambda s : term[@V, @F].\, forall.term_1(\lambda v : @V.\, p(v),\ s), \\
&\qquad\qquad args(t))\ .
\end{aligned}$$

For the two occurrences of $@F$ in the selector types of constructor *apply*, $forall.term_2$ returns the conjunction of

$$\mathtt{ALL}_{@F}(fsym(t), \epsilon) = p(fsym(t))$$

and

$$\begin{aligned}
&\mathtt{ALL}_{list[term[@V,@F]]}(args(t), \mathbf{12}) \\
&= forall.list(\lambda s : term[@V, @F].\, \mathtt{ALL}_{term[@V,@F]}(s, \mathbf{2}),\ args(t)) \\
&= forall.list(\lambda s : term[@V, @F].\, forall.term_2(\lambda f : @F.\, p(f), s) \\
&\qquad\qquad args(t))\ .
\end{aligned}$$

Procedures $forall.term_1$ and $forall.term_2$ are examples of quantification procedures that are defined by second-order recursion. ♢

3.1.2 Properties of *forall.str*

We defer the termination proof for procedures $forall.str_h$ until Chapter 4 (Lemma 4.68 on p. 129) where we have powerful methods at our disposal that allow us to capture the structural recursion scheme. The following lemma shows that procedures $forall.str_h$ compute the expected result.

Lemma 3.5. *Let P be a terminating program and $\tau := str[\tau_1, \ldots, \tau_k]$ be a ground base type. Then for all $\mathsf{x} \in \mathbb{V}(P)_\tau$, $\mathsf{p} \in \mathbb{V}(P)_{\tau_h \to bool}$, $h \in \{1, \ldots, k\}$, and $\pi \in Pos(\tau)$:*

1. $eval_P(forall.str_h(\mathsf{p}, \mathsf{x})) = true \iff$
   $eval_P(\mathsf{p}(\mathsf{z})) = true$ *for all* $\mathsf{z} \in Itm_\tau(\mathsf{x}, h)$

2. $eval_P(\mathtt{ALL}_\tau(\mathsf{x}, \pi)) = true \iff$
   $eval_P(\mathsf{p}(\mathsf{z})) = true$ *for all* $\mathsf{z} \in Itm_\tau(\mathsf{x}, \pi)$

Proof. We prove the statements simultaneously and for all type constructors *str* by induction on x wrt. the well-founded *proper subterm* relation $<_{\mathcal{T}}$. We call the induction hypotheses $(1')$ and $(2')$, respectively.

Let $\mathsf{x} = cons(q_1, \ldots, q_n)$ for some $q_1, \ldots, q_n \in \mathbb{V}(P)$.

1. If $Occ_{@A_h}(cons) = \emptyset$, then $eval_P(forall.str_h(\mathsf{p}, \mathsf{x})) = true$ and the statement is true, because $Itm_\tau(\mathsf{x}, h) = \emptyset$ by Definition 2.56 (p. 42).

   Otherwise $eval_P(forall.str_h(\mathsf{p}, \mathsf{x})) = true$ iff $(\dagger)$ $eval_P(\mathtt{ALL}_{\tau_j}(q_j, \pi)) = true$ for all $(j, \pi) \in Occ_{@A_h}(cons)$. We make a case analysis over π. By definition of ALL, $(\dagger)$ is equivalent to the conjunction of

   $$eval_P(\mathsf{p}(q_j)) = true \text{ for all } (j, \epsilon) \in Occ_{@A_h}(cons) \tag{3.6}$$

   and

   $$\begin{aligned} &eval_P(forall.str'_i(\lambda y : \tau'_i.\, \mathtt{ALL}_{\tau'_i}(y, \pi'),\ q_j)) = true \\ &\text{for all } (j, i\pi') \in Occ_{@A_h}(cons)\,. \end{aligned} \tag{3.7}$$

   By $(1')$, (3.7) is equivalent to

   $$\begin{aligned} &eval_P(\mathtt{ALL}_{\tau'_i}(\mathsf{z}, \pi')) = true \\ &\text{for all } (j, i\pi') \in Occ_{@A_h}(cons) \text{ and all } \mathsf{z} \in Itm(q_j, i)\,. \end{aligned} \tag{3.8}$$

   Using $(2')$, (3.8) is equivalent to

   $$\begin{aligned} &eval_P(\mathsf{p}(\mathsf{z}')) = true \\ &\text{for all } (j, i\pi') \in Occ_{@A_h}(cons), \text{ all } \mathsf{z} \in Itm(q_j, i), \\ &\text{and all } \mathsf{z}' \in Itm(\mathsf{z}, \pi')\,. \end{aligned} \tag{3.9}$$

By Lemma 2.59 (p. 44), (3.9) is equivalent to

$$\begin{array}{l} eval_P(\mathsf{p}(\mathsf{z})) = true \\ \text{for all } (j, i\pi') \in Occ_{@A_h}(cons) \text{ and all } \mathsf{z} \in Itm(q_j, i\pi')\,. \end{array} \tag{3.10}$$

The conjunction of (3.6) and (3.10) is equivalent to

$$eval_P(\mathsf{p}(\mathsf{z})) = true \text{ for all } \mathsf{z} \in Itm_\tau(\mathsf{x}, h) \tag{3.11}$$

by definition of $Itm_\tau(\mathsf{x}, h)$, which proves (1).

2. If $\pi = \epsilon$, then (2) is trivially satisfied.

   If $\pi = i\pi'$, then $eval_P(\mathtt{ALL}_\tau(\mathsf{x}, \pi)) = true$ iff

   $$eval_P\big(forall.str'_i(\lambda y : \tau_i.\, \mathtt{ALL}_{\tau_i}(y, \pi'),\, \mathsf{x})\big) = true \tag{3.12}$$

   by definition of ALL. By (1), this is equivalent to

   $$eval_P(\mathtt{ALL}_{\tau_i}(\mathsf{z}, \pi')) = true \text{ for all } \mathsf{z} \in Itm_\tau(\mathsf{x}, i)\,. \tag{3.13}$$

   Since $\mathsf{z} <_{\mathcal{T}} \mathsf{x}$, we can apply (2′): (3.13) is equivalent to $eval_P(\mathsf{p}(\mathsf{z}')) = true$ for all $\mathsf{z} \in Itm_\tau(\mathsf{x}, i)$ and all $\mathsf{z}' \in Itm_{\tau_i}(\mathsf{z}, \pi')$. By Lemma 2.59, this is equivalent to $eval_P(\mathsf{p}(\mathsf{z})) = true$ for all $\mathsf{z} \in Itm_\tau(\mathsf{x}, i\pi')$ as desired. □

3.2 Quantification Procedures for Second-Order Procedures

For a second-order procedure

```
procedure proc(f : τ_y → τ_f, x : τ_x) : τ_proc <=
assume c_proc; B_proc
```

we sometimes need to know if the values $y : \tau_y$ that function f will be applied to during the evaluation of $proc(f, x)$ satisfy some predicate p. While some second-order procedures such as *map* (cf. Figure 1.3 on p. 6) apply f to all items of x, other second-order procedures do some more sophisticated calculations and may apply f also to other values.

Figure 3.3 shows procedures to compute subtraction and division on natural numbers. If $x \leq y$, $x - y$ yields 0. Procedure "/" assumes that the denominator y is different from 0 and computes $\lfloor \frac{x}{y} \rfloor$. Suppose that for a list $k = e_1 :: \ldots :: e_n :: \varepsilon$ of natural numbers we would like to compute

$$e_1/(e_2/(\ldots/(e_{n-1}/e_n)\ldots))\,.$$

```
procedure [infixl, 10] −(x, y : ℕ) : ℕ <=
if ?0(x)
  then 0
  else if ?0(y)
         then x
         else −(x) − −(y)
       end
end

procedure [infixl, 20] /(x, y : ℕ) : ℕ <=
assume ?+(y);
if y > x
  then 0
  else +((x − y) / y)
end
```

Figure 3.3: Procedures for subtraction and division of natural numbers

Using procedure *foldr* (Figure 1.4 on p. 7), we can write this as

$$foldr(/, 1, k) \ . \tag{3.14}$$

Since procedure "/" may only be applied to pairs (x, y) with $y \neq 0$, we might like to know whether the items of list k are of such a form that the context requirement of procedure "/" is satisfied when evaluating (3.14). So which pairs (x, y) of values will "/" be applied to during the evaluation of (3.14)?

Let us do an example:

$$\begin{aligned} & eval_P\big(foldr(/, 1, 33 :: 12 :: 24 :: 6 :: \varepsilon)\big) \\ = \ & eval_P\big(33/(12/(24/(6/1)))\big) \\ = \ & 11 \end{aligned}$$

In this example, procedure "/" is applied to $(6, 1)$, $(24, 6)$, $(12, 4)$, and $(33, 3)$. The first component of these pairs of values is determined by the items in list k, while the second component is determined by (i) the items in k, (ii) procedure "/", and (iii) the order that *foldr* applies "/" to the list items.

In the following, we describe the synthesis of a procedure

```
procedure forall.foldr(p : @A × @B → bool,
                       f : @A × @B → @B,
                       x : @B, k : list[@A]) : bool
```

that checks if $p(a, b)$ is satisfied for all (a, b) that f is applied to when evaluating $foldr(f, x, k)$. We use this procedure to automatically synthesize a

so-called *context hypothesis* for (3.14) that ensures that the context requirement of procedure "/" is satisfied (see Section 3.5):

$$\mathit{forall.foldr}(\lambda x, y : \mathbb{N}.\, ?^{+}(y),\ /,\ 1,\ k) \tag{3.15}$$

3.2.1 Uniform Synthesis of *forall.proc*

For the sake of readability, we first define quantification procedures for second-order procedures with one first-order parameter and an (optional) second formal parameter. We generalize this definition afterwards.

Definition 3.6 (Quantification procedures for second-order procedures)**.** *For each second-order procedure*

$$\begin{array}{l}\texttt{procedure}\ \mathit{proc}(f : \tau_1 \times \ldots \times \tau_m \to \tau_f,\ x : \tau_x) : \tau_{proc} <= \\ \texttt{assume}\ c_{proc};\ B_{proc}\end{array}$$

the quantification procedure *forall.proc* for *proc is defined by*

$$\begin{array}{l}\texttt{procedure}\ \mathit{forall.proc}(p : \tau_1 \times \ldots \times \tau_m \to \mathit{bool}, \\ \qquad\qquad\qquad\qquad\qquad f : \tau_1 \times \ldots \times \tau_m \to \tau_f, \\ \qquad\qquad\qquad\qquad\qquad x : \tau_x) : \mathit{bool} <= \\ \texttt{ALL}_f(c_{proc}) \wedge \mathit{if}\{c_{proc}, \texttt{ALL}_f(B_{proc}), \mathit{true}\}\end{array}$$

where

$$\begin{array}{l}\texttt{ALL}_f(v) := \mathit{true} \\ \texttt{ALL}_f(f(t_1, \ldots, t_m)) := p(t_1, \ldots, t_m) \wedge \texttt{ALL}_f(t_1) \wedge \ldots \wedge \texttt{ALL}_f(t_m) \\ \texttt{ALL}_f(g(t_1, \ldots, t_n)) := \texttt{ALL}_f(t_1) \wedge \ldots \wedge \texttt{ALL}_f(t_n) \\ \texttt{ALL}_f(h(\lambda \vec{y}.\, t,\ t')) := \mathit{forall.h}(\lambda \vec{y}.\, \texttt{ALL}_f(t),\ \lambda \vec{y}.\, t,\ t') \wedge \texttt{ALL}_f(t') \\ \texttt{ALL}_f(\mathit{if}\{t_1, t_2, t_3\}) := \texttt{ALL}_f(t_1) \wedge \mathit{if}\{t_1, \texttt{ALL}_f(t_2), \texttt{ALL}_f(t_3)\} \\ \texttt{ALL}_f(\mathit{case}\{t_1; t_2, \ldots, t_n\}) := \texttt{ALL}_f(t_1) \wedge \mathit{case}\{t_1; \texttt{ALL}_f(t_2), \ldots, \texttt{ALL}_f(t_n)\} \\ \texttt{ALL}_f(\mathit{let}\{y\ \texttt{:=}\ t_1;\ t_0\}) := \texttt{ALL}_f(t_1) \wedge \mathit{let}\{y\ \texttt{:=}\ t_1;\ \texttt{ALL}_f(t_0)\}\end{array}$$

for any variable v, any first-order function $g \notin \Sigma(P)^{\mathsf{cond}}$, $g \neq f$, and any second-order procedure h (including proc).

Example 3.7. Figure 3.4 shows the resulting quantification procedures for the second-order procedures in Figure 1.3 (p. 6).

- Procedures *forall.map* and *forall.filter* check if $p(x)$ is satisfied for all items x of list k.
- Procedure *forall.every* checks if $p'(e_i)$ is satisfied for all items $e_1, \ldots, e_\nu$ of list $k = e_1 :: \ldots :: e_n :: \varepsilon$, where $\nu \in \{1, \ldots, n\}$ is the smallest index such that $p(e_\nu)$ is *not* satisfied (because *every* does not call $p(e_{\nu+1})$, $\ldots$, $p(e_n)$ in this case). If no such ν exists, then $\nu := n$. ♢

```
procedure forall.map(p : @A → bool,
                     f : @A → @B,
                     k : list[@A]) : bool <=
if ?ε(k)
  then true
  else if p(hd(k))
         then forall.map(p, f, tl(k))
         else false
       end
end

procedure forall.every(p′, p : @A → bool,
                       k : list[@A]) : bool <=
if ?ε(k)
  then true
  else if p′(hd(k))
         then if p(hd(k))
                then forall.every(p′, p, tl(k))
                else true
              end
         else false
       end
end

procedure forall.filter(p′, p : @A → bool,
                        k : list[@A]) : bool <=
if ?ε(k)
  then true
  else if p′(hd(k))
         then forall.filter(p′, p, tl(k))
         else false
       end
end
```

Figure 3.4: Automatically synthesized quantification procedures for procedures *map*, *every*, and *filter*, cf. Figure 1.3 (p. 6)

```
procedure forall.foldl(p : @A × @B → bool,
                       f : @A × @B → @A,
                       x : @A, k : list[@B]) : bool <=
if ?ε(k)
  then true
  else if p(x, hd(k))
         then forall.foldl(p, f, f(x, hd(k)), tl(k))
         else false
       end
end

procedure forall.foldr(p : @A × @B → bool,
                       f : @A × @B → @B,
                       x : @B, k : list[@A]) : bool <=
if ?ε(k)
  then true
  else if p(hd(k), foldr(f, x, tl(k)))
         then forall.foldr(p, f, x, tl(k))
         else false
       end
end

procedure forall.rev_itlist(p : @A × @B → bool,
                            f : @A × @B → @B,
                            x : @B, k : list[@A]) : bool <=
if ?ε(k)
  then true
  else if p(hd(k), x)
         then forall.rev_itlist(p, f, f(hd(k), x), tl(k))
         else false
       end
end
```

Figure 3.5: Automatically synthesized quantification procedures for the *fold* procedures of Figure 1.4 (p. 7)

Example 3.8. Figure 3.5 shows the resulting quantification procedures for the second-order procedures in Figure 1.4 (p. 7). ♢

Definition 3.6 is generalized to second-order procedures of arbitrary arity as follows: Let *proc* be a second-order procedure that is defined by

```
procedure proc(x_1 : τ_1, ..., x_n : τ_n) : τ_proc <=
assume c_proc; B_proc .
```

For each $i \in \{1, \ldots, n\}$ such that τ_i is a function type $\tau_i = \tau'_1 \times \ldots \times \tau'_m \to \tau'_{m+1}$, quantification procedure

$$\begin{aligned}&\texttt{procedure}\ \mathit{forall.proc}_i(p : \tau'_1 \times \ldots \times \tau'_m \to \mathit{bool},\\ &\qquad\qquad\qquad\qquad x_1 : \tau_1, \ldots, x_n : \tau_n) : \mathit{bool} <=\\ &\texttt{ALL}_{x_i}(c_{proc}) \wedge \mathit{if}\{c_{proc}, \texttt{ALL}_{x_i}(B_{proc}), \mathit{true}\}\end{aligned}$$

is defined as in Definition 3.6. The straightforward generalization of the definition of $\texttt{ALL}_f\big(h(\lambda\vec{y}.\, t,\, t')\big)$ to arbitrary numbers of first-order arguments $\lambda\vec{y}_1.\, t_1, \ldots, \lambda\vec{y}_k.\, t_k$ and arguments $t'_1, \ldots, t'_n$ of base types is:

$$\begin{aligned}&\texttt{ALL}_{x_i}\big(h(\lambda\vec{y}_1.\, t_1, \ldots, \lambda\vec{y}_k.\, t_k,\ t'_1, \ldots, t'_n)\big) :=\\ &\qquad\bigwedge_{\nu=1}^{k} \mathit{forall.h}_\nu\big(\lambda\vec{y}_\nu.\, \texttt{ALL}_{x_i}(t_\nu),\ \lambda\vec{y}_1.\, t_1, \ldots, \lambda\vec{y}_k.\, t_k,\ t'_1, \ldots, t'_n\big)\\ &\quad\wedge \texttt{ALL}_{x_i}(t'_1) \wedge \ldots \wedge \texttt{ALL}_{x_i}(t'_n)\end{aligned}$$

3.2.2 Properties of *forall.proc*

In this section we show that procedure *forall.proc* computes the expected result. We expect that $\mathit{forall.proc}(\mathsf{p}, \mathsf{f}, \mathsf{x})$ yields *true* iff $\mathsf{p}(\vec{q})$ yields *true* for all $\vec{q}$ with $\mathit{proc}(\mathsf{f}, \mathsf{x}) \rhd_{\mathsf{f}} \mathsf{f}(\vec{q})$.

However, we need to phrase our expectation more precisely. For instance, consider the procedure call $\mathit{map}(\mathit{hd}, (0 :: \varepsilon) :: \varepsilon)$. We have:

1. $\mathit{map}(\mathit{hd}, (0 :: \varepsilon) :: \varepsilon) \rhd_{\mathit{hd}} \mathit{hd}((0 :: \varepsilon) :: \varepsilon)$

2. $\mathit{map}(\mathit{hd}, (0 :: \varepsilon) :: \varepsilon) \rhd_{\mathit{hd}} \mathit{hd}(0 :: \varepsilon)$

We get *two* calls of *hd*, although list $(0 :: \varepsilon) :: \varepsilon$ contains just *one* element $(0 :: \varepsilon)$. In fact, procedure *forall.map* only checks $\mathsf{p}(0 :: \varepsilon)$, which corresponds to the second call of *hd*. This second call of *hd* stems from the instantiation of parameter f with *hd*, whereas the first call of *hd* occurs already in the body of *map*.

Therefore the following lemma demands that f be a *fresh function*. This means that f is a λ-expression $\mathsf{f} = \lambda\vec{y}.\, g(\vec{y})$ such that function symbol g neither occurs in the body of *proc* nor in the bodies of the procedures *proc*′ with $\mathit{proc} >^+_{\mathit{uses}} \mathit{proc}'$. Alternatively, one can imagine f as uniquely labeled (e. g., as $\underline{\mathsf{f}}$) so that in the example above we can tell the calls of *hd* apart:

1. $\mathit{map}(\underline{\mathit{hd}}, (0 :: \varepsilon) :: \varepsilon) \not\rhd_{\underline{\mathit{hd}}} \mathit{hd}((0 :: \varepsilon) :: \varepsilon)$, because $\mathit{hd} \neq \underline{\mathit{hd}}$

2. $\mathit{map}(\underline{\mathit{hd}}, (0 :: \varepsilon) :: \varepsilon) \rhd_{\underline{\mathit{hd}}} \underline{\mathit{hd}}(0 :: \varepsilon)$

Lemma 3.9. *Let P be a terminating program. Then for all second-order procedures $proc \in \Sigma(P)$ as in Definition 3.6, all grounding type substitutions $\theta \in GndSubst_{\Omega(P)}(\tau_1, \ldots, \tau_m, \tau_f, \tau_x)$, all values $\mathsf{x} \in \mathbb{V}(P)_{\theta(\tau_x)}$, and all fresh functions $\mathsf{p} \in \mathbb{V}(P)_{\theta(\tau_1 \times \ldots \times \tau_m \to bool)}$ and $\mathsf{f} \in \mathbb{V}(P)_{\theta(\tau_1 \times \ldots \times \tau_m \to \tau_f)}$:*

1. $eval_P(forall.proc(\mathsf{p}, \mathsf{f}, \mathsf{x})) \in \{true, false\}$
2. $forall.proc(\mathsf{p}, \mathsf{f}, \mathsf{x}) \rhd_{\mathsf{f}} \mathsf{f}(q_1, \ldots, q_m) \implies proc(\mathsf{f}, \mathsf{x}) \rhd_{\mathsf{f}} \mathsf{f}(q_1, \ldots, q_m)$
3. $forall.proc(\mathsf{p}, \mathsf{f}, \mathsf{x}) \rhd_{\mathsf{p}} \mathsf{p}(q_1, \ldots, q_m) \implies proc(\mathsf{f}, \mathsf{x}) \rhd_{\mathsf{f}} \mathsf{f}(q_1, \ldots, q_m)$
4. $eval_P(forall.proc(\mathsf{p}, \mathsf{f}, \mathsf{x})) = true \iff$
   $eval_P(\mathsf{p}(q_1, \ldots, q_m)) = true$ *for all* $q_1, \ldots, q_m \in \mathbb{V}(P)$
   with $proc(\mathsf{f}, \mathsf{x}) \rhd_{\mathsf{f}} \mathsf{f}(q_1, \ldots, q_m)$

Proof. The idea is to show the claims via a proof by contradiction (which actually is an inductive proof). Suppose that there were a $>_{uses}$-minimal procedure *proc* and some $\succ^{\theta}_{proc}$-minimal values (f, x) such that one of the claims was violated for some p.

Claims (1)–(3) hold because of the following invariant of $\texttt{ALL}_f$ (to be proved by structural induction on term t): If the resulting term $t' := \texttt{ALL}_f(t)$ contains a function call $\varphi(t_1, \ldots, t_n)$ under call context C', then at least one of the following holds:

(i) t contains a function call $\varphi(t_1, \ldots, t_n)$ under some call context $C \subseteq C'$.

(ii) $\varphi(t_1, \ldots, t_n) = p(t_1, \ldots, t_m)$ and t contains a function call $f(t_1, \ldots, t_m)$ under some call context $C \subseteq C'$.

(iii) $\varphi(t_1, \ldots, t_n) = forall.h(t_1, t_2, t_3)$ and t contains a function call $h(t_2, t_3)$ under some call context $C \subseteq C'$.

This invariant entails (1) termination of *forall.proc* (because *forall.proc* contains no more recursive calls than *proc* by (i) and (iii)) as well as claims (2) (by (i) and (iii)) and (3) of the lemma (by (ii) and (iii)). Claim (4) cannot be violated either because of the following invariant of $\texttt{ALL}_f$ (again to be proved by structural induction on term t):

$$eval_P(\texttt{ALL}_f(t)[p/\mathsf{p}, f/\mathsf{f}, x/\mathsf{x}]) = true \iff$$
$$eval_P(\mathsf{p}(q_1, \ldots, q_m)) = true \text{ for all } q_1, \ldots, q_m \in \mathbb{V}(P)$$
$$\text{with } \mathsf{f}(q_1, \ldots, q_m) \in Calls_{\mathsf{f}}(t[p/\mathsf{p}, f/\mathsf{f}, x/\mathsf{x}])$$

Thus there cannot exist minimal *proc*, f, and x that violate the claims, which proves the lemma. □

Checking properties of function calls. The construction $\mathtt{ALL}_f(t)$ of Definition 3.6 checks if some predicate p holds for all f-calls. We can generalize this construction in a straightforward way to arbitrary many functions $f_1, \ldots, f_m$ to check that each call $f_j(q_1, \ldots, q_{n_j})$ satisfies a certain predicate $p_{f_j}(q_1, \ldots, q_{n_j})$.

Let P be a terminating $\mathcal{L}$-program and let $t \in \mathcal{T}(\Sigma(P), \mathcal{V})$ be a normalized term of a base type. Furthermore, let

- $\mathcal{V}_0 \subseteq \mathcal{V}_\mathsf{f}(t)$ be the set of all term variables of base types in t,
- $\mathcal{V}_1 \subseteq \mathcal{V}_\mathsf{f}(t)$ be the set of all first-order term variables in t,
- $\Sigma_1 \subseteq \Sigma(t) \setminus \Sigma(P)^{\mathsf{cond}}$ be the set of all first-order function symbols in t different from *if* and *case*, and
- $\Sigma_2 \subseteq \Sigma(t)$ be the set of all second-order function symbols in t.

For each function type $\tau = \tau_1 \times \ldots \times \tau_n \to \tau_{n+1} \in \mathit{Types}(\Omega(P), \mathcal{W})$ and each $f \in \mathcal{V}_1 \cup \Sigma_1 \cup \Sigma_2$ with $f\!:\tau$, let $p_f : \tau_1 \times \ldots \times \tau_n \to \mathit{bool}$ be a fresh term variable.

We define a term $\mathit{Chk}(t) \in \mathcal{T}(\Sigma(P), \mathcal{V} \cup \{p_f \mid f \in \mathcal{V}_1 \cup \Sigma_1 \cup \Sigma_2\})_{\mathit{bool}}$ that checks if p_f is satisfied for each call of f in t by case analysis over the form of term t. In the following definition, $x \in \mathcal{V}_0$ is a term variable, $f \in \mathcal{V}_1 \cup \Sigma_1$ is a first-order function or term variable, and $h \in \Sigma_2$ is a second-order procedure:

$$
\begin{aligned}
&\mathit{Chk}(x) := \mathit{true}\\
&\mathit{Chk}(f(t_1, \ldots, t_n)) := p_f(t_1, \ldots, t_n) \wedge \mathit{Chk}(t_1) \wedge \ldots \wedge \mathit{Chk}(t_n)\\
&\mathit{Chk}(h(\lambda\vec{y}.\, t_1,\ t_2)) := p_h(\lambda\vec{y}.\, t_1,\ t_2) \wedge \mathit{Chk}(t_2) \wedge\\
&\qquad\qquad\qquad \mathit{forall}.h(\lambda\vec{y}.\, \mathit{Chk}(t_1),\ \lambda\vec{y}.\, t_1,\ t_2)\\
&\mathit{Chk}(\mathit{if}\{t_1, t_2, t_3\}) := \mathit{Chk}(t_1) \wedge \mathit{if}\{t_1, \mathit{Chk}(t_2), \mathit{Chk}(t_3)\}\\
&\mathit{Chk}(\mathit{case}\{t_1; t_2, \ldots, t_n\}) := \mathit{Chk}(t_1) \wedge \mathit{case}\{t_1; \mathit{Chk}(t_2), \ldots, \mathit{Chk}(t_n)\}\\
&\mathit{Chk}(\mathit{let}\{x := t_1;\ t_0\}) := \mathit{Chk}(t_1) \wedge \mathit{let}\{y := t_1;\ \mathit{Chk}(t_0)\}
\end{aligned}
$$

This construction appears in various guises for different purposes. For instance, in Section 3.5 we use it to generate so-called *context hypotheses* that ensure that for each function call $f(q_1, \ldots, q_n)$ in a term t, the context requirement $c_f[q_1, \ldots, q_n]$ of f is satisfied.

3.3 Existential Quantification

In the previous sections we have defined quantification procedures $\mathit{forall}.\mathit{str}_h$ and $\mathit{forall}.\mathit{proc}_i$ for finite universal quantification. Sometimes we also need finite existential quantification. To simplify the notation in these cases, we define $\mathit{exists}.\mathit{str}_h$ and $\mathit{exists}.\mathit{proc}_i$ as abbreviation for negated universal quantification:

Definition 3.10 (Existential quantification procedures)**.** *For each definition of a data structure* $str[@A_1, \ldots, @A_k]$ *with* $k \geq 1$*, procedure* $exists.str_h$ *is defined by*

$$\begin{array}{l}\texttt{procedure}\ exists.str_h(p : @A_h \to bool,\\ \qquad\qquad\qquad\qquad x : str[@A_1, \ldots, @A_k]) : bool <=\\ \neg\, forall.str_h(\lambda z : @A_h.\, \neg\, p(z),\ x)\,.\end{array}$$

For a procedure

$$\begin{array}{l}\texttt{procedure}\ proc(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau_{proc} <=\\ \texttt{assume}\ c_{proc};\ B_{proc}\end{array}$$

and each $i \in \{1, \ldots, n\}$ *such that* τ_i *is a function type* $\tau_i = \tau'_1 \times \ldots \times \tau'_m \to \tau'_{m+1}$*, procedure* $exists.proc_i$ *is defined by*

$$\begin{array}{l}\texttt{procedure}\ exists.proc_i(p : \tau'_1 \times \ldots \times \tau'_m \to bool,\\ \qquad\qquad\qquad\qquad x_1 : \tau_1, \ldots, x_n : \tau_n) : bool <=\\ \neg\, forall.proc_i(\lambda z_1 : \tau'_1, \ldots, z_m : \tau'_m.\, \neg\, p(z_1, \ldots, z_m),\ x_1, \ldots, x_n)\,.\end{array}$$

Existential quantification procedures are never actually synthesized in the implementation. Instead, we always use the universal quantification procedures as in Definition 3.10. In order to simplify subsequent proofs, we briefly state and prove the characterizing properties of $exists.str_h$ and $exists.proc_i$:

Lemma 3.11. *Let* P *be a terminating program and* $\tau := str[\tau_1, \ldots, \tau_k]$ *be a ground base type. Then for all* $\mathsf{x} \in \mathbb{V}(P)_\tau$*,* $\mathsf{p} \in \mathbb{V}(P)_{\tau_h \to bool}$*,* $h \in \{1, \ldots, k\}$*, and* $\pi \in Pos(\tau)$*:*

$$\begin{array}{l}eval_P(exists.str_h(\mathsf{p}, \mathsf{x})) = true \iff\\ eval_P(\mathsf{p}(\mathsf{z})) = true \text{ for some } \mathsf{z} \in Itm_\tau(\mathsf{x}, h)\end{array}$$

Proof. The claim follows from Lemma 3.5 (p. 68). □

Lemma 3.12. *Let* P *be a terminating program. Then for all second-order procedures* $proc \in \Sigma(P)$ *as in Definition 3.6 (p. 71), all grounding type substitutions* $\theta \in GndSubst_{\Omega(P)}(\tau_1, \ldots, \tau_m, \tau_f, \tau_x)$*, all values* $\mathsf{x} \in \mathbb{V}(P)_{\theta(\tau_x)}$*, and all fresh functions* $\mathsf{p} \in \mathbb{V}(P)_{\theta(\tau_1 \times \ldots \times \tau_m \to bool)}$ *and* $\mathsf{f} \in \mathbb{V}(P)_{\theta(\tau_1 \times \ldots \times \tau_m \to \tau_f)}$*:*

$$\begin{array}{l}eval_P(exists.proc(\mathsf{p}, \mathsf{f}, \mathsf{x})) = true \iff\\ eval_P(\mathsf{p}(q_1, \ldots, q_m)) = true \text{ for some } q_1, \ldots, q_m \in \mathbb{V}(P)\\ \text{with } proc(\mathsf{f}, \mathsf{x}) \rhd_{\mathsf{f}} \mathsf{f}(q_1, \ldots, q_m)\end{array}$$

Proof. The claim follows from Lemma 3.9 (p. 75). □

Example 3.13. The existential quantification procedures for the second-order procedures in Figure 1.3 (p. 6) compute the following:

- Procedures *exists.map* and *exists.filter* check if $p(x)$ is satisfied for some item x of list k.
- Procedure *exists.every* checks if $p'(e_i)$ is satisfied for at least one of the items $e_1, \ldots, e_\nu$ of list $k = e_1 :: \ldots :: e_n :: \varepsilon$, where $\nu \in \{1, \ldots, n\}$ is the smallest index such that $p(e_\nu)$ is *not* satisfied. If no such ν exists, then $\nu := n$. ♢

3.4 Usage of Quantification Procedures

Now that we have introduced quantification procedures for data structures and second-order procedures, we quickly summarize in which contexts they are useful. Quantification procedures are used

1. in context hypotheses (see Section 3.5),
2. in termination hypotheses (see Section 4.1), and
3. in induction hypotheses (see Sections 5.2 and 5.3).

We write Σ^{all} for the signature of all quantification procedures $forall.str_h$ and $forall.proc_i$ in Σ and we write Σ^{ex} for the signature of all "procedures" $exists.str_h$ and $exists.proc_i$ in Σ.

3.5 Context Correctness

Each function symbol f is associated with a context requirement c_f. If this context requirement is violated, the interpreter returns a witness value, cf. Section 2.3.1. This may lead to unexpected results, because one usually assumes that all context requirements are satisfied when thinking about a formula.

In order to prevent such errors, ✓`eriFun` synthesizes *context hypotheses* for procedures and lemmas [72]. These context hypotheses need to be proved and ensure that the context requirements of all functions that occur in a procedure body or in the formula of a lemma are satisfied. Consequently, the cases where a function call is evaluated to a witness value (cf. Section 2.3) do not occur in practice.

We generally define the context requirement of a term t as follows (this definition is an instance of the generic construction $Chk(t)$ described on p. 76; it extends the definition from [72] to indirect function calls):

Definition 3.14 (Context requirement of a term)**.** *The* context requirement $CR(t) \in \mathcal{T}(\Sigma, \mathcal{V})_{bool}$ *of a normalized term* $t \in \mathcal{T}(\Sigma, \mathcal{V})$ *is defined by*

$$
\begin{aligned}
&CR(x) := \mathit{true}\\
&CR(f(t_1, \ldots, t_n)) := c_f[t_1, \ldots, t_n] \wedge CR(t_1) \wedge \ldots \wedge CR(t_n)\\
&CR(g(\lambda\vec{x}.\, t,\ t')) := c_g[\lambda\vec{x}.\, t,\ t'] \wedge \mathit{forall}.g(\lambda\vec{x}.\, CR(t),\ \lambda\vec{x}.\, t,\ t') \wedge CR(t')\\
&CR(\mathit{if}\{t_1, t_2, t_3\}) := CR(t_1) \wedge \mathit{if}\{t_1, CR(t_2), CR(t_3)\}\\
&CR(\mathit{case}\{t_1; t_2, \ldots, t_n\}) := CR(t_1) \wedge \mathit{case}\{t_1; CR(t_2), \ldots, CR(t_n)\}\\
&CR(\mathit{let}\{x \texttt{ := } t_1;\ t_0\}) := CR(t_1) \wedge \mathit{let}\{x \texttt{ := } t_1;\ CR(t_0)\}\\
&CR(\lambda\vec{x}.\, t) := \mathit{true}
\end{aligned}
$$

for any variable x*, any first-order function* $f \notin \Sigma(P)^{\mathsf{cond}}$*, and any second-order procedure* g*.*[4]

Example 3.15. The context requirement for term $\mathit{map}(^-, k)$ is given by $\mathit{forall}.\mathit{map}(?^+, ^-, k)$. This Boolean term expresses that no element of list k is `0`, so the predecessor function $^-(\ldots)$ can be safely applied to all elements of list k. ♢

Example 3.16. The context requirement for term $\mathit{foldr}(/, n, k)$ is

$$CR\big(\mathit{foldr}(/, n, k)\big) = \mathit{forall}.\mathit{foldr}(\lambda x, y : \mathbb{N}.\, ?^+(y),\ /, n, k)\ .$$

This Boolean term expresses that upon evaluation of $\mathit{foldr}(/, n, k)$ no division by zero occurs. ♢

Definition 3.17 (Context hypotheses)**.** *Let proc be a procedure that is defined by*

$$\texttt{procedure}\ \mathit{proc}(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \mathrel{<=} \texttt{assume}\ c_{proc};\ B_{proc}\ .$$

The context hypothesis for *proc is defined as*

$$\mathit{CtxHyp}_{proc} := \forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ CR(c_{proc}) \wedge \big(c_{proc} \rightarrow CR(B_{proc})\big)\,.$$

The context hypothesis for a lemma

$$\texttt{lemma}\ \mathit{lem} \mathrel{<=} \forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ b$$

is defined as

$$\mathit{CtxHyp}_{lem} := \forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ CR(b)\,.$$

We say that procedure proc is context correct *iff* CtxHyp_{proc} *is true. Similarly, lemma* lem *is* context correct *iff* CtxHyp_{lem} *is true.*

[4] Case $g(\lambda\vec{x}.\, t,\ t')$ is generalized in a straightforward way to second-order procedures with more than one first-order parameter by using the corresponding procedures $\mathit{forall}.g_i$.

```
procedure forall.forall.map_1(p', p : @A → bool,
                                f : @A → @B,
                                k : list[@A]) : bool <=
if ?ε(k)
  then true
  else if p'(hd(k))
         then if p(hd(k))
                then forall.forall.map_1(p', p, f, tl(k))
                else true
              end
         else false
       end
end
```

Figure 3.6: Quantification procedure $forall.forall.map_1$

3.6 Limiting the Usage and Synthesis of Quantification Procedures

As a rule of thumb, quantification procedure *forall.proc* is required whenever procedure *proc* occurs in a user-defined expression (e. g., in a procedure or lemma definition). So what do we do if quantification procedure *forall.proc* occurs in a user-defined expression? Do we need to synthesize quantification procedures *forall.forall.proc*, *forall.forall.forall.proc*, ...?

In general, the answer is "yes": If the user writes $map(^-, k)$, we synthesize context hypothesis $forall.map(?^+, ^-, k)$. If the user writes a term like $forall.map(prime, dbl, k)$ for a procedure

```
procedure prime(n : ℕ) : bool <=
assume ¬ 2 > n; ...
```

then the context hypothesis for this procedure call requires quantification procedure $forall.forall.map_1$ (see Figure 3.6):

$$forall.forall.map_1(\lambda n : \mathbb{N}.\, \neg\, 2 > n,\ prime, dbl, k)$$

This context hypothesis ensures that $forall.map(prime, dbl, k)$ applies *prime* only to natural numbers $n \geq 2$.

This small example is

- complicated wrt. the resulting context hypothesis, because it becomes difficult to remember that $forall.forall.map_1(p', p, f, k)$ checks if $p'(x)$ is satisfied for all calls $p(x)$ that are necessary to evaluate procedure call $forall.map(p, f, k)$;

- artificial, because it is unlikely that the user writes terms such as $forall.map(prime, dbl, k)$ in his definitions.

Since it is generally unlikely that the user finds the quantification procedures *forall.proc* helpful for his own definitions (and that is why the example above is artificial), we can easily limit the usage and the synthesis of quantification procedures.

Limiting the usage of quantification procedures. Quantification procedures $forall.str_h$ are easily understandable and generally useful. For instance, procedure *groundterm* (Figure 1.5 on p. 9) can be implemented using the automatically synthesized quantification procedure *forall.list* instead of the (semantically equivalent) user-defined second-order procedure *every*. Also, procedure *subterm* can be implemented using *forall.list* and negation instead of the user-defined second-order procedure *some* (which is semantically equivalent to *exists.list*). Therefore, the user should be allowed to use these procedures $forall.str_h$ wherever he wishes.

Quantification procedures *forall.proc* for a user-defined second-order procedure *proc* or a quantification procedure $proc = forall.str$ are required to reason about function calls and thus serve a purely technical purpose. Therefore they should not be called in user-defined procedures. However, the user may need to formulate and prove an auxiliary lemma about *forall.proc* if *forall.proc* occurs in a context or termination hypothesis. Consequently, *forall.proc* may be used in lemma definitions.

Limiting the synthesis of quantification procedures. For each data structure $str[@A_1, \ldots, @A_k]$ and each $h = 1, \ldots, k$, we synthesize quantification procedures $forall.str_h$ and $forall.forall.str_h$.

Additionally, for each second-order procedure $proc : \tau_1 \times \ldots \times \tau_n \to \tau$ that is *not* a quantification procedure, a quantification procedure $forall.proc_i$ is synthesized for each $i = 1, \ldots, n$ such that τ_i is a function type.

No further quantification procedures are synthesized.

Justification and consequences. Since $forall.str_h$ may be used without restriction, we need to synthesize $forall.forall.str_h$. Similarly, a user-defined second-order procedure *proc* may be used without restriction, so quantification procedure *forall.proc* needs to be synthesized (we omit index i in $forall.proc_i$ from now on).

The question that remains to be answered is: Why can we stop the synthesis of quantification procedures at this level? After all, *forall.proc* may occur in user-defined lemmas so that we require quantification procedure *forall.forall.proc* for the context hypothesis of this lemma. We avoid the synthesis of *forall.forall.proc* by *approximating* it safely using *forall.proc* as explained below.

Approximation of *forall.forall.proc*. For a second-order procedure

$$\texttt{procedure}\ \mathit{proc}(f : \tau_1 \times \ldots \times \tau_m \to \tau_f,\ x : \tau_x) : \tau_{\mathit{proc}}$$

the quantification procedure

$$\begin{aligned}\texttt{procedure}\ \mathit{forall.proc}(p : \tau_1 \times \ldots \times \tau_m \to \mathit{bool},\\ f : \tau_1 \times \ldots \times \tau_m \to \tau_f,\\ x : \tau_x) : \mathit{bool}\end{aligned}$$

checks if $p(q_1, \ldots, q_m)$ is satisfied for all calls $f(q_1, \ldots, q_m)$ by procedure *proc*. According to Lemma 3.9 (p. 75), *forall.proc* calls $p(q_1, \ldots, q_m)$ only if *proc* calls $f(q_1, \ldots, q_m)$. Moreover, *forall.proc* calls $f(q_1, \ldots, q_m)$ only if *proc* calls $f(q_1, \ldots, q_m)$. Hence both $\mathit{forall.forall.proc}_1(p', p, f, x)$—i. e., checking p' for the p-calls by *forall.proc*—and $\mathit{forall.forall.proc}_2(p', p, f, x)$—i. e., checking p' for the f-calls of *forall.proc*—can be approximated by $\mathit{forall.proc}(p', f, x)$.

This approximation is safe in the sense that the approximation implies $\mathit{forall.forall.proc}_i(p', p, f, x)$. Thus context hypotheses and termination hypotheses are strengthened, meaning that the user might need to prove a stronger hypothesis. In general, this strengthening might render a hypothesis unprovable. In practice, however, we observed that strengthening makes it easier to prove a hypothesis, because known (or easy to find) lemmas about *forall.proc* can be used, whereas lemmas about $\mathit{forall.forall.proc}_i$ would have to be discovered and proved otherwise.

Example 3.18. Procedure $\mathit{forall.forall.map}_1$ (cf. Figure 3.6) can be approximated by using $\mathit{forall.map}(p', f, k)$ instead of $\mathit{forall.forall.map}_1(p', p, f, k)$.

Thus the context hypothesis for procedure call $\mathit{forall.map}(\mathit{prime}, \mathit{dbl}, k)$ can be approximated by $\mathit{forall.map}(\lambda n : \mathbb{N}. \neg\, 2 > n,\ \mathit{dbl}, k)$. This context hypothesis is stronger than the original context hypothesis, because it requires *all* elements of list k to be ≥ 2, whereas the original context hypothesis only requires that the first elements of list k up to the first element that is not a prime number are ≥ 2. ♢

3.7 Summary

Finite quantification $\forall z \in M.\ p(z)$ for a decidable predicate p is decidable. To exploit this decidability in automated theorem proving, we synthesize so-called *quantification procedures* as decision procedures for finite quantification:

- For a data structure $str[@A_1, \ldots, @A_k]$, $forall.str_h(p, x)$ yields *true* iff predicate $p : @A_h \to bool$ is satisfied for all items $z : @A_h$ in $x : str[@A_1, \ldots, @A_k]$.
- For a second-order procedure

  $$\texttt{procedure}\ proc(f : \tau_1 \times \ldots \times \tau_m \to \tau_f,\ x : \tau_x) : \tau_{proc}$$

  $forall.proc(p, f, x)$ yields *true* iff predicate $p : \tau_1 \times \ldots \times \tau_m \to bool$ is satisfied for all arguments $z_1, \ldots, z_m$ that *proc* calls f with, i. e., for all $z_1, \ldots, z_m$ with $proc(p, x) \rhd_f f(z_1, \ldots, z_m)$.

  If procedure *proc* has more than one first-order parameter f, we synthesize a quantification procedure for each first-order parameter: Procedure $forall.proc_i$ then denotes the quantification procedure for the i-th parameter of *proc*.

Since we sometimes also need decision procedures for finite existential quantification, we define *exists.str* and *exists.proc* as abbreviation for negated universal quantification based on *forall.str* and *forall.proc*. Although they are never actually synthesized, we call them "procedures".

In order to avoid that procedures *forall.forall.proc* need to be synthesized, we restrict the use of *forall.proc* by the user. Theoretically, the synthesis of quantification procedures *forall.forall.proc* is no problem, but it becomes difficult to understand their meaning so that we choose to prevent their occurrence in proof obligations that the user needs to consider.

Chapter 4

Termination Analysis

In an $\mathcal{L}$-program, procedures are defined successively in such a way that each procedure may only call itself or a procedure that has already been defined. This means that *mutual recursion* is not allowed so that we can split up the termination analysis of a whole program into separate termination analysis problems, one for each procedure. By Lemma 2.83 (p. 58), a procedure f terminates iff the recursive call relation $\succ_f^\theta$ is well-founded for each grounding type substitution θ.

A widely used technique to show that a procedure $f : \tau_1 \times \ldots \times \tau_n \to \tau$ terminates is to provide a so-called *measure function* $m_\theta : \mathbb{V}(P)_{\theta(\tau_1)} \times \ldots \times \mathbb{V}(P)_{\theta(\tau_n)} \to \mathbb{N}$ that is uniformly defined for each grounding type substitution θ such that $m_\theta(q_1, \ldots, q_n)$ strictly decreases for each recursive call; thus $m_\theta(q_1, \ldots, q_n)$ is an upper bound on the number of recursive calls that are required to evaluate $f(q_1, \ldots, q_n)$.[1] The following lemma formally states this principle. According to Definition 2.80 (p. 57), we only need to consider values $q_i \in \mathbb{V}(P \setminus \{f\})_{\theta(\tau_i)}$ that do not call procedure f in a λ-expression.

Lemma 4.1. *Let P be an arbitrary $\mathcal{L}$-program that contains a procedure $f : \tau_1 \times \ldots \times \tau_n \to \tau$. Procedure f terminates if*

1. *all procedures $g \in P \setminus \{f\}$ with $f >^+_{uses} g$ terminate and*
2. *for each grounding type substitution $\theta \in GndSubst_{\Omega(P)}(\tau_1, \ldots, \tau_n)$ there exists a function $m_\theta : \mathbb{V}(P)_{\theta(\tau_1)} \times \ldots \times \mathbb{V}(P)_{\theta(\tau_n)} \to \mathbb{N}$ such that for all $q_1, \ldots, q_n \in \mathbb{V}(P \setminus \{f\})$ with $q_i \in \mathbb{V}(P \setminus \{f\})_{\theta(\tau_i)}$ for $i = 1, \ldots, n$:*

$$m_\theta(q_1, \ldots, q_n) > m_\theta(q'_1, \ldots, q'_n)$$
$$\text{for all } q'_1, \ldots, q'_n \in \mathbb{V}(P) \text{ with } (q_1, \ldots, q_n) \succ_f^\theta (q'_1, \ldots, q'_n)\,.$$

[1] This technique is used in the theorem provers ACL2 [58], Isabelle [66], PVS [69], and √eriFun [91], for example. It can be regarded as an instance of the *size-change termination principle* [63]: A program terminates if every infinite call sequence would cause an infinite descent in some well-founded data value. This descent is given by the values of the measure function.

Proof. If procedure f did not terminate, then there were a grounding type substitution θ and values $q_1, \ldots, q_n \in \mathbb{V}(P \setminus \{f\})$ with

$$(q_1, \ldots, q_n) \succ_f^\theta (q'_1, \ldots, q'_n) \succ_f^\theta (q''_1, \ldots, q''_n) \succ_f^\theta \ldots$$

for some q'_i, q''_i. But then

$$m_\theta(q_1, \ldots, q_n) > m_\theta(q'_1, \ldots, q'_n) > m_\theta(q''_1, \ldots, q''_n) > \ldots$$

was an infinite descending sequence of natural numbers, which is impossible. Thus procedure f terminates. □

Proving termination of a procedure using Lemma 4.1 mainly involves

- finding an appropriate measure function m_θ and
- showing that the measure function satisfies requirement (2) of the lemma.

Requirement (1) of Lemma 4.1 is a simple book-keeping process: Termination of the procedures in an $\mathcal{L}$-program P is analyzed in the order of their definition. Once termination of a procedure g has been proved, g gets a flag "terminating". Checking requirement (1) then means checking if all procedures g with $f >^+_{uses} g$ have flag "terminating".

Requirement (2) of Lemma 4.1 is formally represented by so-called *termination hypotheses*, which are universally quantified formulas of the form $\forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ b$.

Organization of this chapter. In interactive termination analysis, it is the user's responsibility to specify an appropriate measure function. The task of the theorem prover is then to generate termination hypotheses which entail that requirement (2) of Lemma 4.1 is satisfied. We describe the generation process in Section 4.1.

Section 4.2 lays the foundation of automated termination analysis of procedures with second-order recursion. It introduces an extension of Walther's approach of *argument-bounded functions* [86, 96]. The approach is based on uniformly defined measure functions that measure the "structural size" of values (e. g., the length of a list). Section 4.3 presents a calculus that automates a significant part of the proof of requirement (2) of Lemma 4.1. It properly subsumes the calculus from [86, 96] and additionally solves some termination problems that the original approach could not cope with.

Section 4.4 introduces a new concept to reason about second-order recursion: *call-boundedness*. A call-bounded second-order procedure calls its first-order parameter only with values of a bounded structural size. Virtually all common second-order procedures enjoy this property. Using call-boundedness of second-order procedures, we show how termination of procedures with second-order recursion can by analyzed automatically in Section 4.5.

4.1 Interactive Termination Analysis

In order to show termination of a procedure

$$\texttt{procedure}\ \mathit{proc}(x_1:\tau_1,\ldots,x_n:\tau_n):\tau <= \texttt{assume}\ c_{proc};\ B_{proc}$$

the user can specify a *measure term* $\mathtt{m} \in \mathcal{T}(\Sigma(P),\{x_1,\ldots,x_n\})_{\mathbb{N}}$ from which we uniformly define measure functions m_θ by

$$m_\theta(q_1,\ldots,q_n) := \mathit{eval}_P(\mathtt{m}[q_1,\ldots,q_n])$$

for each $\theta \in \mathit{GndSubst}_{\Omega(P)}(\tau_1,\ldots,\tau_n)$ and all values $q_1,\ldots,q_n \in \mathbb{V}(P)$ with $q_i \in \mathbb{V}(P)_{\theta(\tau_i)}$ for $i = 1,\ldots,n$.[2]

For each position $\pi \in \Pi^{rec}_{proc}(B^{\mathsf{rel}}_{proc})$ of a recursive call we define a *termination hypothesis* $\mathit{TermHyp}_{\mathtt{m}}(B^{\mathsf{rel}}_{proc},\pi)$ that ensures that measure $\mathtt{m}$ decreases for the arguments of the recursive call.

Let t be a *let*-free subterm of B^{rel}_{proc} and $\pi \in \Pi^{rec}_{proc}(t)$. We distinguish two cases:

- If $\pi \in \Pi^{rec}_{proc}(t) \cap \mathit{TLPos}(t)$, then term position π denotes a *direct* recursive call $t|_\pi = \mathit{proc}(t_1,\ldots,t_n)$. This recursive call is evaluated iff the conditions of the call context $\mathit{COND}(t,\pi)$ are satisfied. In this case, the measure needs to decrease:

$$\mathit{TermHyp}_{\mathtt{m}}(t,\pi) := \bigwedge \mathit{COND}(t,\pi) \rightarrow \mathtt{m}[x_1,\ldots,x_n] > \mathtt{m}[t_1,\ldots,t_n]$$

- If $\pi \in \Pi^{rec}_{proc}(t) \setminus \mathit{TLPos}(t)$, then term position π denotes an *indirect* recursive call and procedure *proc* is defined by second-order recursion. Hence there is a minimal prefix $\pi' \in \mathit{TLPos}(t)$ of π with $t|_{\pi'} = h(\lambda\vec{y}.\, t'',\, t')$ for a second-order procedure h; i. e., $\pi = \pi'\mathbf{10}\pi''$ for some $\pi'' \in \mathit{Pos}(t'')$.[3]

  The call of the second-order procedure h is evaluated iff the conditions of the call context $\mathit{COND}(t,\pi')$ are satisfied. Thus we check if the measure decreases for the recursive call at position π'' in t'':

$$\begin{aligned}\mathit{TermHyp}_{\mathtt{m}}(t,\pi) := \bigwedge \mathit{COND}(t,\pi') \rightarrow\ & \\ \mathit{forall}.h(\lambda\vec{y}.\, \mathit{TermHyp}_{\mathtt{m}}(t'',\pi''),\ \lambda\vec{y}.\, t'',\ t')&\end{aligned}$$

[2] Strictly speaking, eval_P returns a constructor ground term $q \in \mathbb{V}(P)_{\mathbb{N}}$ that *represents* a natural number, but *is not* a natural number. This can be easily remedied by using a function $\iota : \mathbb{V}(P)_{\mathbb{N}} \rightarrow \mathbb{N}$ that translates each object-level natural number q into the corresponding natural number n.

[3] Recall that $h(\lambda\vec{y}.\, t'',\, t')|_{\mathbf{10}} = (\lambda\vec{y}.\, t'')|_{\mathbf{0}} = t''$.

Definition 4.2 (Termination hypotheses wrt. measure term)**.** *For a procedure*

$$\texttt{procedure}\ proc(x_1:\tau_1,\ldots,x_n:\tau_n):\tau <= B^{\mathsf{rel}}_{proc}$$

and a measure term $\mathtt{m} \in \mathcal{T}(\Sigma(P),\{x_1,\ldots,x_n\})_{\mathbb{N}}$*, the* termination hypotheses for *proc* wrt. $\mathtt{m}$ *are given by*

$$th^{\pi}_{proc,\mathtt{m}} := \forall x_1:\tau_1,\ldots,x_n:\tau_n.\ TermHyp_{\mathtt{m}}(B^{\mathsf{rel}}_{proc},\pi)$$

for each $\pi \in \Pi^{rec}_{proc}(B^{\mathsf{rel}}_{proc})$*.*

Theorem 4.3. *A procedure*

$$\texttt{procedure}\ proc(x_1:\tau_1,\ldots,x_n:\tau_n):\tau <= B^{\mathsf{rel}}_{proc}$$

of an $\mathcal{L}$*-program* P *terminates if all procedures* $g \in P$ *with* $f >^{+}_{uses} g$ *terminate and if there exists a measure term* $\mathtt{m} \in \mathcal{T}(\Sigma(P),\{x_1,\ldots,x_n\})_{\mathbb{N}}$ *such that for each* $\pi \in \Pi^{rec}_{proc}(B^{\mathsf{rel}}_{proc})$ *termination hypothesis* $th^{\pi}_{proc,\mathtt{m}}$ *is true.*

Proof. Procedure f terminates, because the requirements of Lemma 4.1 are satisfied: The conjunction of the termination hypotheses is semantically equivalent to the instantiation of $Chk(B^{\mathsf{rel}}_{proc})$ (see p. 76) with[4]

$$p_{proc} := \lambda x'_1:\tau_1,\ldots,x'_n:\tau_n.\,\mathtt{m}[x_1,\ldots,x_n] > \mathtt{m}[x'_1,\ldots,x'_n]\,.$$

i. e., $Chk(B^{\mathsf{rel}}_{proc}) \approx \bigwedge_{\pi\in\Pi^{rec}_{proc}(B^{\mathsf{rel}}_{proc})} th^{\pi}_{proc,\mathtt{m}}$.

Since $eval_P(\mathsf{x}>\mathsf{y}) = true$ iff $\mathsf{x} > \mathsf{y}$ for all $\mathsf{x},\mathsf{y} \in \mathbb{V}(P)_{\mathbb{N}}$ (cf. the definition of procedure ">" in Figure 2.2 on p. 38), the termination hypotheses entail that $m_\theta(q_1,\ldots,q_n) > m_\theta(q'_1,\ldots,q'_n)$ for all $q'_1,\ldots,q'_n \in \mathbb{V}(P)$ with $(q_1,\ldots,q_n) \succ^{\theta}_{f} (q'_1,\ldots,q'_n)$. □

Example 4.4. Procedure *map* (cf. Figure 1.3 on p. 6) terminates: For measure term $\mathtt{m} := |\,k\,|$ we get the single termination hypothesis

$$th^{\mathbf{32}}_{map,\mathtt{m}} = \forall f:@A \to @B, k:list[@A].\ if\{\neg\,?\varepsilon(k), |\,k\,| > |\,tl(k)\,|, true\}\,,$$

which is easily provable. ♢

[4]Definition 4.2 splits up $Chk(B^{\mathsf{rel}}_{proc})$ into separate termination hypotheses for each recursive call, because it is simpler to prove these smaller formulas individually than proving the conjunction in one sweep.

```
procedure termsize(t : term[@V, @F]) : ℕ <=
case t of
  var   : 1,
  apply : foldl(+, 1, map(termsize, args(t)))
end
```

Figure 4.1: Procedure *termsize* to compute the size of a term

Example 4.5. Procedure *groundterm* (cf. Figure 1.5 on p. 9) terminates: For measure term $\mathtt{m} := termsize(t)$, cf. Figure 4.1, termination hypothesis

$$
\begin{aligned}
&th^{\mathbf{310}}_{groundterm,\mathtt{m}} = \\
&\quad \forall t : term[@V, @F]. \\
&\quad if\{?apply(t), \\
&\qquad forall.every(\lambda s : term[@V, @F].\, termsize(t) > termsize(s), \\
&\qquad\qquad groundterm, \\
&\qquad\qquad args(t)), \\
&\qquad true\}
\end{aligned}
$$

is true (procedure *termsize* computes the number of variable symbols and function symbols in a term t). The proof of the termination hypothesis requires some auxiliary lemmas. Section 4.5 shows how this termination proof is performed fully automatically. ♢

Note that the termination hypothesis of procedure *groundterm* in Example 4.5 contains function symbol *groundterm*. Since termination of procedure *groundterm* has not been proved yet, we may not use the definition of *groundterm* in the proof of its termination hypothesis. The fact that a procedure *proc* terminates regardless of the semantics of *proc* is called *strong termination* [86, 87].

In order to prove strong termination of a procedure *proc*, any information about the semantics of *proc* that is relevant to prove termination of *proc* needs to be encoded in the conditions of a recursive call. For instance, strong termination of procedure

```
procedure foo(n : ℕ) : ℕ <=
if ?0(n)
  then 0
  else if n > foo(⁻(n))
         then foo(foo(⁻(n)))
         else 0
       end
end
```

can be shown, because condition $n > foo(^-(n))$ ensures that the argument of the outer recursive call in $foo(foo(^-(n)))$ is smaller than n. Without this condition, procedure *foo* does not terminate strongly.

4.2 Automated Termination Analysis

The method of *argument-bounded functions* [86, 96] has been developed to prove termination of first-order procedures. The approach can also be used to prove termination of a second-order procedure if the procedure does not contain indirect recursive calls. The following example illustrates the idea behind the approach.

Example 4.6. Termination of *every* (cf. Figure 1.3 on p. 6) can be easily proved using the method of argument-bounded functions: Selector *tl* is *argument-bounded*, which intuitively means $\#(k) \geq \#(tl(k))$ for all lists $k \neq \varepsilon$, where $\#(k)$ counts the occurrences of *list*-constructors ε and :: in k (and thus corresponds to the length of list k plus 1). A system-generated *difference procedure* [86, 96] $\Delta_{tl} : list[@A] \to bool$ decides if this inequality is strict for a given list k, which is the case if $k \neq \varepsilon$. To prove that the second argument of procedure *every* gets strictly smaller in the recursive call $every(p, tl(k))$, it suffices to show the termination hypothesis $\forall k : list[@A].\ k \neq \varepsilon \wedge p(hd(k)) \to \Delta_{tl}(k)$, which is trivial to prove. ♢

Proving termination of a procedure that is defined by second-order recursion (e. g., *groundterm* in Figure 1.5 on p. 9), however, is challenging and hence is the main problem we tackle. The key observation is that *every* applies p only to members x of list k. While in Isabelle the user needs to state and prove this knowledge explicitly as a *congruence theorem*, our approach *automatically* extracts such information from the definition of *every*.

More specifically, our approach detects that for any instantiation of $@A$ with a type τ, the number of τ-constructors in each value $x : \tau$ that p is applied to by *every* is bounded by the number of τ-constructors in the elements e of list k: $\sum_{e \in k} \#_\tau(e) \geq \#_\tau(x)$. We say that *every* is *call-bounded* wrt. p. For the second-order recursion in procedure *groundterm* and $args(t) = t_1 :: \ldots :: t_n :: \varepsilon$ this means $\#_{term}(t_1) + \ldots + \#_{term}(t_n) \geq \#_{term}(x)$. Since $t = apply(fsym(t), args(t))$ contains one *term*-constructor more than $args(t)$, we have $\#_{term}(t) > \#_{term}(t_1) + \ldots + \#_{term}(t_n) \geq \#_{term}(x)$, so *groundterm* is only called recursively with arguments x that are smaller than t, which ensures termination of *groundterm*.

Formally, we parameterize the size measure # by a type position so that for $args(t) : list[term[@V, @F]]$ we can separately count the *list*- and *term*-constructors. This allows us to consider selector $args : term[@V, @F] \to list[term[@V, @F]]$ as argument-bounded wrt. type component $term[\ldots]$ (i. e., $args(t)$ contains no more *term*-constructors than t), and the difference procedure Δ_{args} returns *true* iff $?apply(t)$ holds.

4.2.1 A Uniform Size Measure

Our size measure $\#_\tau(t,\pi)$ for terms $t:\tau$ is parameterized with a type position $\pi \in Pos(\tau)$ so that we can precisely specify which data constructors are to be counted. The computation of the size is very similar to the computation of the items of a term (cf. Definition 2.56 on p. 42 and Lemma 4.11 below).

Intuitively, the size $\#_\tau(t,\pi)$ of a term $t \in \mathcal{T}(\Sigma(P)^{\mathsf{c}})_\tau$ is computed as follows: We replicate the type (and data) constructor definitions so that each type constructor occurs at most once in type τ. Then $\#_\tau(t,\pi)$ counts the $\tau\|_\pi$-constructors in t. For example, $list[list[\mathbb{N}]]$ is transformed into $\tau := listA[listB[\mathbb{N}]]$, so $\#_\tau(t,\epsilon)$ counts the $listA$-constructors in t (because $\tau\|_\epsilon = listA$) and $\#(t,\mathbf{1})$ counts the $listB$-constructors in t (because $\tau\|_{\mathbf{1}} = listB$).

The formal definition of the size measure below directly uses the type position without needing to replicate any type constructors. It is based on a data structure definition of the form

$$\begin{array}{l}\texttt{structure}\ str[@A_1,\ldots,@A_k] <= \\ \quad \ldots, \\ \quad cons(sel_1:\tau_1,\ldots,sel_n:\tau_n), \\ \quad \ldots\end{array}$$

Definition 4.7 (Size measure)**.** *For each ground base type $\tau = str[\tau'_1,\ldots,\tau'_k]$ the* size measure $\#_\tau : \mathcal{T}(\Sigma^{\mathsf{c}})_\tau \times Pos(\tau) \to \mathbb{N}$ *is defined by*

$$\#_\tau(cons(t_1,\ldots,t_n),\pi) := \begin{cases} 1 & \text{if } \pi=\epsilon \text{ and } cons \in \mathcal{C}^{\mathsf{irr}}_{str}, \\ 2+\sum_{(j,\pi')\in Occ_{str}(cons)} \#_{\theta(\tau_j)}(t_j,\pi') & \text{if } \pi=\epsilon \text{ and } cons \in \mathcal{C}^{\mathsf{refl}}_{str}, \\ \sum_{(j,\pi')\in Occ_{@A_h}(cons)} \#_{\theta(\tau_j)}(t_j,\pi'\pi'') & \text{if } \pi = h\pi'', \end{cases}$$

where $\theta := \{@A_1/\tau'_1,\ldots,@A_k/\tau'_k\}$ instantiates the type variables of str. If type τ is obvious from the context, we will usually omit the type index in $\#_\tau$.

Irreflexive data constructors get weight 1. A reflexive data constructor *cons* in a term $cons(t_1,\ldots,t_n)$ is counted with weight[5] 2 and we recurse into those $t_j : \theta(\tau_j)$ that *by definition of cons* may also contain *str*-constructors ($\tau_j\|_{\pi'} = str$). For instance, for $\tau := list[list[\mathbb{N}]]$ and $\pi := \epsilon$ we recursively add the size $\#_\tau(t_2,\epsilon)$ of the *tl*-component of $t_1 :: t_2$, whereas we do *not* recurse into the *hd*-component t_1. Finally, for $\pi = h\pi''$ we recursively add up the sizes of those t_j that contain $\tau'_h\|_{\pi''}$-constructors, so we recurse into the occurrences of the h-th type parameter $@A_h$ in τ_j. For example, for $\tau := list[list[\mathbb{N}]]$, term $t_1 :: t_2$, and $\pi := \mathbf{1}$, $\#_{list[\mathbb{N}]}(t_1,\epsilon) + \#_{list[list[\mathbb{N}]]}(t_2,\mathbf{1})$ counts the *list*-constructors of the *inner* lists.

[5] This simplifies some size estimation proofs; e. g., one can prove that $apply(f,l)$ is greater than $var(v)$ without having to check if the argument list l is non-empty.

Definition 4.7 generalizes the size measure of [86], which considers only case $\pi = \epsilon$. The following examples instantiate Definition 4.7 with various types. We discuss alternatives to this definition of the size measure in Section 4.2.5.

Example 4.8. For type $\tau := \mathit{list}[\mathbb{N}]$, $\#_{\mathit{list}[\mathbb{N}]}(t, \epsilon) = 2R+I$ for the numbers R and I of occurrences of :: and ε in t, respectively, so $R = |\,t\,|$ and $I = 1$. $\#_{\mathit{list}[\mathbb{N}]}(t, \mathbf{1})$ is the sum of the sizes of the elements in list t and thus is equal to $\sum_{n \in t} \#_{\mathbb{N}}(n, \epsilon) = \sum_{n \in t}(2n + 1)$. ♢

Note that $\#_{\mathit{list}[\mathbb{N}]}(\varepsilon, \mathbf{1}) = 0$, whereas $\#_{\mathit{list}[\mathbb{N}]}(\mathtt{0} :: \varepsilon, \mathbf{1}) = \#_{\mathbb{N}}(\mathtt{0}, \epsilon) = 1 \neq 0$. Thus $\#_{\mathit{list}[\mathbb{N}]}(t, \mathbf{1}) = 0$ iff $t = \varepsilon$. This a useful property, cf. Examples 4.14 (p. 93) and 4.24 (p. 99).

Example 4.9. For type $\tau := \mathit{pair}[\mathbb{N}, \mathbb{N}]$, $\#(x, \mathbf{1}) = 2R+I$ for the numbers R and I of occurrences of ${}^{+}(\ldots)$ and $\mathtt{0}$ in $\mathit{fst}(x)$, respectively. Similarly, $\#(x, \mathbf{2})$ counts the occurrences of ${}^{+}(\ldots)$ and $\mathtt{0}$ in $\mathit{snd}(x)$. ♢

Example 4.10. For type $\tau := \mathit{term}[\tau_V, \tau_F]$, where τ_V and τ_F are arbitrary ground base types, $\#_{\mathit{term}[\tau_V, \tau_F]}(t, \epsilon)$ counts the occurrences of *term*-constructors *var* (with weight 1) and *apply* (with weight 2) in t. ♢

The following lemma relates the size of a term t with the items of t:

Lemma 4.11. *For each ground base type $\tau = \mathit{str}[\tau'_1, \ldots, \tau'_k]$, each type position $\pi := \pi_1\pi_2 \in \mathit{Pos}(\tau)$, and each term $t \in \mathcal{T}(\Sigma^{\mathsf{c}})_\tau$ we have*

$$\#(t, \pi_1\pi_2) = \sum_{t' \in \mathit{Itm}(t, \pi_1)} \#(t', \pi_2) \ .$$

Proof. If $\pi_1 = \epsilon$, then $\mathit{Itm}(t, \epsilon) = \{\!|t|\!\}$, so $\#(t, \pi_2) = \sum_{t' \in \mathit{Itm}(t, \epsilon)} \#(t', \pi_2)$. If $\pi_1 \neq \epsilon$, we prove the statement by structural induction on t.

$t = \mathit{cons}$: $\#(t, \pi_1\pi_2) = 0 = \sum_{t' \in \mathit{Itm}(t, \pi_1)} \#(t', \pi_2)$, because $\mathit{Itm}(t, \pi_1) = \emptyset$.

$t = \mathit{cons}(t_1, \ldots, t_n)$: Let $\pi_1 = h\pi''$ for some $h \in \mathbb{N}$ and $\pi'' \in \mathbb{N}^*$. The induction hypothesis is

$$\text{(IH)} \quad \#(t_i, \pi'_1\pi'_2) = \textstyle\sum_{t' \in \mathit{Itm}(t_i, \pi'_1)} \#(t', \pi'_2)$$

for all $i \in \{1, \ldots, n\}$, π'_1, and π'_2. Hence:

$$\begin{aligned}
&\#(t, h\pi''\pi_2) \\
&= \textstyle\sum_{(j,\pi') \in \mathit{Occ}_{@A_h}(\mathit{cons})} \#(t_j, \pi'\pi''\pi_2) && \text{; by def. of } \# \\
&= \textstyle\sum_{(j,\pi') \in \mathit{Occ}_{@A_h}(\mathit{cons})} \sum_{t' \in \mathit{Itm}(t_j, \pi'\pi'')} \#(t', \pi_2) && \text{; by (IH)} \\
&= \textstyle\sum_{t' \in \mathit{Itm}(t, h\pi'')} \#(t', \pi_2) && \text{; by def. of } \mathit{Itm}
\end{aligned}$$

as desired. □

For the synthesis of difference procedures for selectors (see Section 4.2.3) we need to determine in which cases $\#(t,\pi)$ is greater than zero (i. e., if term t contains an item at type position π). Therefore we define a predicate $\#^{\geqslant 1}(t,\pi)$ that decides if $\#(t,\pi) \geq 1$:

Definition 4.12 ($\#^{\geqslant 1}_\tau(t,\pi)$)**.** *For each base type $\tau = str[\tau'_1,\ldots,\tau'_k]$, each type position $\pi \in Pos(\tau)$, and each term $t \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$, term $\#^{\geqslant 1}_\tau(t,\pi) \in \mathcal{T}(\Sigma,\mathcal{V})_{bool}$ is defined by*

$$\#^{\geqslant 1}_\tau(t,\pi) := \begin{cases} true & \text{if } \pi = \epsilon, \\ exists.str_h\big(\lambda y : \tau'_h.\, \#^{\geqslant 1}_{\tau'_h}(y,\pi'),\ t\big) & \text{if } \pi = h\pi'\,. \end{cases}$$

Lemma 4.13. *For each base type $\tau = str[\tau'_1,\ldots,\tau'_k]$, each type position $\pi \in Pos(\tau)$, each grounding type substitution $\theta \in GndSubst_{\Omega(P)}(\tau)$, and each value $q \in \mathbb{V}(P)_{\theta(\tau)}$:*

$$eval_P\big(\#^{\geqslant 1}_\tau(q,\pi)\big) = true \iff \#_{\theta(\tau)}(q,\pi) \geq 1$$

Proof. We prove the lemma by induction on q wrt. the well-founded *proper subterm* relation $<_{\mathcal{T}}$.

If $\pi = \epsilon$, then $eval_P(\#^{\geqslant 1}_\tau(q,\epsilon)) = true$ and $\#_{\theta(\tau)}(q,\epsilon) \geq 1$.

If $\pi = h\pi'$, then

$$\begin{aligned}
& eval_P(\#^{\geqslant 1}_\tau(q,h\pi')) = true \\
\iff\ & eval_P(exists.str_h(\lambda y : \tau'_h.\, \#^{\geqslant 1}_{\tau'_h}(y,\pi'),\ q)) = true && \text{; by Def. 4.12} \\
\iff\ & eval_P(\#^{\geqslant 1}_{\tau'_h}(q',\pi')) = true \text{ for some } q' \in Itm(q,h) && \text{; by Lem. 3.11} \\
\iff\ & \#(q',\pi') \geq 1 \text{ for some } q' \in Itm(q,h) && \text{; by (IH)} \\
\iff\ & \#_{\theta(\tau)}(q,h\pi') \geq 1 && \text{; by Lem. 4.11}
\end{aligned}$$

as desired. □

Example 4.14. For $\tau := list[@A]$ and $k \in \mathcal{T}(\Sigma,\mathcal{V})_{list[@A]}$ we get

$$\begin{aligned}
& \#^{\geqslant 1}_{list[@A]}(k,\mathbf{1}) \\
=\ & exists.list(\lambda y : @A.\, true,\ k) \\
=\ & \neg\, forall.list(\lambda y : @A.\, false,\ k) \\
\approx\ & \neg\,\big(?\varepsilon(k) \vee (false \wedge forall.list(\lambda y : @A.\, false,\ tl(k)))\big) \\
\approx\ & \neg\, ?\varepsilon(k)
\end{aligned}$$

The simplified term $\neg\, ?\varepsilon(k)$ can be obtained automatically by symbolic evaluation of $exists.list(\lambda y : @A.\, true,\ k)$. The idea is to unfold the definition of procedure *forall.list* (cf. Figure 3.1 on p. 66) and to simplify the resulting term using $(false \wedge a) \approx false$ and $(a \vee false) \approx a$. See Section 5.4 for details on symbolic evaluation. ♢

Example 4.15. For $\tau := list[term[@\,V, @F]]$ and $k \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$ we get

$$\begin{aligned}
&\#^{\geqslant 1}_{list[term[@\,V,@F]]}(k, \mathbf{12}) \\
&= exists.list(\lambda t : term[@\,V, @F].\, \#^{\geqslant 1}_{term[@\,V,@F]}(t, \mathbf{2}),\ k) \\
&= exists.list(\lambda t : term[@\,V, @F].\, exists.term_2(\lambda f : @F.\, true,\ t),\ k) \\
&\approx \neg\, forall.list(\lambda t : term[@\,V, @F].\, forall.term_2(\lambda f : @F.\, false,\ t),\ k) \\
&\approx \neg\, forall.list(\lambda t : term[@\,V, @F].\, ?var(t),\ k)
\end{aligned}$$

The result corresponds to the intuition that a list k of terms contains at least one function symbol $f : @F$ iff not all terms in list k are variables. ♢

4.2.2 Argument-Bounded Functions

A function f is called *argument-bounded* iff the result $f(\ldots, t, \ldots)$ of a function call is bounded by argument t of the call wrt. the size measure (provided that the function may be applied to t); e. g., $\#(tl(k), \epsilon) \leq \#(k, \epsilon)$ for each $k \not= \varepsilon$. Such facts are used to show that some parameter x of a procedure p decreases in recursive calls if f is used in the argument of a recursive call. For instance, we used argument-boundedness of tl to show termination of procedure *every* in Example 4.6 (p. 90).

For the sake of readability we consider *unary* functions f at first and generalize the definitions in Section 4.2.4.

Definition 4.16 (Argument-bounded functions)**.** *A function $f : \tau \to \tau'$ with context requirement c_f is* (π, ϱ)-argument-bounded *for type positions $\pi \in Pos(\tau)$ and $\varrho \in Pos(\tau')$ iff*

1. *τ is a base type with $\tau\|_\pi = \tau'\|_\varrho$ and*
2. *for all grounding type substitutions $\theta \in GndSubst_{\Omega(P)}(\tau)$ and all values $q \in \mathbb{V}(P)_{\theta(\tau)}$ with $eval_P(c_f[q]) = true$:*

$$\#_{\theta(\tau)}(q, \pi) \geq \#_{\theta(\tau')}(eval_P(f(q)), \varrho)\,.$$

Requirement (1) ensures that we compare the sizes wrt. the same type constructor. Requirement (2) basically states that argument q is greater than or equal to the return value of f (provided that f may be applied to this input), where the size is measured wrt. the type constructor agreed upon in (1). This definition generalizes the corresponding definition of [86, 96], which considers only the case $\pi = \varrho = \epsilon$.

Example 4.17. Procedure *last* (cf. Figure 4.2) is $(\mathbf{1}, \epsilon)$-argument-bounded: The size of the last element of list k is bounded by the sum of the sizes of k's elements. ♢

```
procedure last(k : list[@A]) : @A <=
assume ?::(k);
if ?ε(tl(k))
  then hd(k)
  else last(tl(k))
end

procedure split(k : list[@A]) : pair[list[@A], list[@A]] <=
if ?ε(k)
  then (ε • ε)
  else if ?ε(tl(k))
         then (k • ε)
         else let s := split(tl(tl(k))) in
                (hd(k) :: fst(s) • hd(tl(k)) :: snd(s))
end    end    end
```

Figure 4.2: Procedures *last* and *split*

Example 4.18. Procedure *split* (cf. Figure 4.2) implements the *divide* step of Mergesort and is both $(\epsilon, \mathbf{1})$- and $(\epsilon, \mathbf{2})$-argument-bounded: It splits a list $e_1 :: e_2 :: e_3 :: e_4 :: \ldots$ into lists $e_1 :: e_3 :: \ldots$ and $e_2 :: e_4 :: \ldots$, which are not longer than the input list. ♢

Selectors are argument-bounded, as they return a component of their input:

Theorem 4.19. *Let $sel_j : \tau \to \tau_j$ be a selector, $\tau = str[@A_1, \ldots, @A_k]$, $\pi \in Pos(\tau)$, and $\varrho \in Pos(\tau_j)$. If $\tau\|_\pi = \tau_j\|_\varrho$, then sel_j is (π, ϱ)-argument-bounded.*

Proof. We show that the requirements of Definition 4.16 are satisfied:

1. τ is a base type with $\tau\|_\pi = \tau_j\|_\varrho$.
2. Let $cons : \tau_1 \times \ldots \times \tau_n \to \tau$ be the *str*-constructor that sel_j belongs to and let $q \in \mathbb{V}(P)_{\theta(\tau)}$. Then $eval_P(c_{sel_j}[q]) = true$ entails $q = cons(q_1, \ldots, q_n)$ for some $q_i \in \mathbb{V}(P)_{\theta(\tau_i)}$ by the definition of c_{sel_j}, see Definition 2.32 (p. 31).

   We perform a case analysis over $\pi \in Pos(\tau) = \{\epsilon, 1, \ldots, k\}$:

   Case $\pi = \epsilon$: Since $\tau_j\|_\varrho = \tau\|_\epsilon = str$ implies (†) $(j, \varrho) \in Occ_{str}(cons)$, data constructor *cons* is reflexive. Hence:

$$\begin{aligned} \#(q, \epsilon) &= 2 + \textstyle\sum_{(j', \pi') \in Occ_{str}(cons)} \#(q_{j'}, \pi') && \text{; def. of } \# \\ &\geq 2 + \#(q_j, \varrho) && \text{; by (†)} \\ &> \#(q_j, \varrho) = \#(eval_P(sel_j(q)), \varrho) && \text{; def. of } eval_P \end{aligned}$$

Case $\pi \neq \epsilon$: Then $\pi = h$ for some $h \in \{1, \ldots, k\}$, and $\tau_j\|_\varrho = \tau\|_h = @A_h$ implies (†) $(j, \varrho) \in Occ_{@A_h}(cons)$. Hence:

$$\begin{aligned} \#(q, \pi) &= \textstyle\sum_{(j',\pi') \in Occ_{@A_h}(cons)} \#(q_{j'}, \pi') && \text{; def. of } \# \\ &\geq \#(q_j, \varrho) && \text{; by (†)} \\ &= \#(eval_P(sel_j(q)), \varrho) && \text{; def. of } eval_P \end{aligned}$$

Thus sel_j is indeed (π, ϱ)-argument-bounded. □

Example 4.20. The following selectors of Figure 2.1 (p. 31) are argument-bounded:

- $^-(\ldots) : \mathbb{N} \to \mathbb{N}$ is (ϵ, ϵ)-argument-bounded.
- $hd : list[@A] \to @A$ is $(\mathbf{1}, \epsilon)$-argument-bounded: The size of the first element of a non-empty list k is bounded by the sum of the sizes of all elements in k.
- $tl : list[@A] \to list[@A]$ is (ϵ, ϵ)-argument-bounded, because $tl(k)$ contains fewer *list*-constructors "::" than k.

  Selector tl is also $(\mathbf{1}, \mathbf{1})$-argument-bounded, because $tl(k)$ contains a subset of the elements in k.
- $fst : pair[@A, @B] \to @A$ is $(\mathbf{1}, \epsilon)$-argument-bounded.
- $snd : pair[@A, @B] \to @B$ is $(\mathbf{2}, \epsilon)$-argument-bounded.
- $vsym : term[@V, @F] \to @V$ is $(\mathbf{1}, \epsilon)$-argument-bounded.
- $fsym : term[@V, @F] \to @V$ is $(\mathbf{2}, \epsilon)$-argument-bounded.
- $args : term[@V, @F] \to list[term[@V, @F]]$ is $(\epsilon, \mathbf{1})$-argument-bounded, because $args(t)$ contains fewer *term*-constructors *var* and *apply* than t.

  Selector *args* is also $(\mathbf{1}, \mathbf{11})$-argument-bounded, because $args(t)$ contains no more variables than t.

  Finally, selector *args* is $(\mathbf{2}, \mathbf{12})$-argument-bounded, because $args(t)$ contains no more function symbols than t. ♢

4.2.3 Difference Functions

Using argument-bounded functions, we can establish inequalities such as $\#(k, \epsilon) \geq \#(tl(k), \epsilon)$ to ensure that the second argument of procedure *every* does not increase in the recursive call. However, this inequality needs to be strict to guarantee termination of *every*. Strictness of such inequalities is expressed by so-called *difference procedures*; e. g., $\Delta^{1,\epsilon}_{tl,\epsilon} : list[@A] \to bool$ returns *true* iff $\#(k, \epsilon) > \#(tl(k), \epsilon)$.

Definition 4.21 (Difference function)**.** *For a (π, ϱ)-argument-bounded function $f : \tau \to \tau'$ with context requirement c_f, $\Delta^{1,\pi}_{f,\varrho} : \tau \to bool$ is a (π, ϱ)-*difference function[6] *for f iff $\Delta^{1,\pi}_{f,\varrho}$ also has context requirement c_f and for all type substitutions $\theta \in GndSubst_{\Omega(P)}(\tau)$ and all values $q \in \mathbb{V}(P)_{\theta(\tau)}$ with $eval_P(c_f[q]) = true$:*

1. $eval_P\big(\Delta^{1,\pi}_{f,\varrho}(q)\big) \in \{true, false\}$ *and*

2. $eval_P\big(\Delta^{1,\pi}_{f,\varrho}(q)\big) = true \iff \#_{\theta(\tau)}(q, \pi) > \#_{\theta(\tau')}(eval_P(f(q)), \varrho)$.

(ϵ, ϱ)-argument-bounded selectors have quite simple difference procedures, because the selector cancels the leading data constructor, cf. Figure 4.3(a):

Theorem 4.22. *Let $sel_j : \tau \to \tau_j$ be an (ϵ, ϱ)-argument-bounded selector for some $\varrho \in Pos(\tau_j)$. Then an (ϵ, ϱ)-difference procedure for sel_j is given by*

$$\texttt{procedure}\ \Delta^{1,\epsilon}_{sel_j,\varrho}(x : \tau) : bool <= \texttt{assume}\ ?cons(x);\quad true$$

for the str-constructor cons that sel_j belongs to.

Proof. Let $q \in \mathbb{V}(P)_{\theta(\tau)}$ such that $eval_P(c_{sel_j}[q]) = true$. Hence $q = cons(q_1, \ldots, q_n)$ for some $q_i \in \mathbb{V}(P)_{\theta(\tau_i)}$.

Thus $eval_P(\Delta^{1,\epsilon}_{sel_j,\varrho}(q)) = true$, and indeed $\#(q, \epsilon) > \#(eval_P(sel_j(q)), \varrho)$ for the same reason as in the proof of Theorem 4.19. □

The synthesis of (π, ϱ)-difference procedures for selectors with $\pi \neq \epsilon$ is a bit more involved. Since $\pi \in Pos(str[@A_1, \ldots, @A_k])$, we have $\pi = h$ for some $h \in \{1, \ldots, k\}$. The idea behind the following theorem is to rearrange the sum

$$\#(cons(q_1, \ldots, q_n), h) = \sum_{(j', \pi') \in Occ_{@A_h}(cons)} \#(q_{j'}, \pi')$$

into the form $\#(q_j, \varrho) + X$. Then $\#(eval_P(sel_j(q)), \varrho) = \#(q_j, \varrho) < \#(q, h)$ for $q = cons(q_1, \ldots, q_n)$ iff $X \geq 1$.

For example, $\Delta^{1,\mathbf{1}}_{hd,\epsilon}(a :: k)$ is supposed to return *true* iff $\#(a :: k, \mathbf{1}) > \#(a, \epsilon)$. Since $\#(a :: k, \mathbf{1}) = \#(a, \epsilon) + \#(k, \mathbf{1})$, this inequality holds iff $\#(k, \mathbf{1}) \geq 1$. The latter can be easily checked using $\#^{\geqslant 1}(k, \mathbf{1})$.

[6] The upper index "1" in $\Delta^{1,\pi}_{f,\varrho}$ designates the *first* parameter of f. We already include it here to be consistent with the generalization described in Section 4.2.4.

(a) $\texttt{procedure}\ \Delta^{1,\epsilon}_{-_{(\ldots)},\epsilon}(x:\mathbb{N}) : \mathit{bool} <= \texttt{assume}\ ?^{+}(x);\ \ \mathit{true}$
$\texttt{procedure}\ \Delta^{1,\epsilon}_{tl,\epsilon}(k:\mathit{list}[@A]) : \mathit{bool} <= \texttt{assume}\ ?::(k);\ \ \mathit{true}$
$\texttt{procedure}\ \Delta^{1,\epsilon}_{\mathit{args},\mathbf{1}}(t:\mathit{term}[@V,@F]) : \mathit{bool} <= \texttt{assume}\ ?\mathit{apply}(t);\ \mathit{true}$

(b) $\texttt{procedure}\ \Delta^{1,\mathbf{1}}_{hd,\epsilon}(k:\mathit{list}[@A]) : \mathit{bool} <= \texttt{assume}\ ?::(k);\ \ \neg ?\varepsilon(\mathit{tl}(k))$
$\texttt{procedure}\ \Delta^{1,\mathbf{1}}_{tl,\mathbf{1}}(k:\mathit{list}[@A]) : \mathit{bool} <= \texttt{assume}\ ?::(k);\ \ \mathit{true}$
$\texttt{procedure}\ \Delta^{1,\mathbf{1}}_{fst,\epsilon}(x:\mathit{pair}[@A,@B]) : \mathit{bool} <= \mathit{false}$
$\texttt{procedure}\ \Delta^{1,\mathbf{2}}_{snd,\epsilon}(x:\mathit{pair}[@A,@B]) : \mathit{bool} <= \mathit{false}$
$\texttt{procedure}\ \Delta^{1,\mathbf{1}}_{\mathit{vsym},\epsilon}(t:\mathit{term}[@V,@F]) : \mathit{bool} <= \texttt{assume}\ ?\mathit{var}(t);\ \mathit{false}$
$\texttt{procedure}\ \Delta^{1,\mathbf{2}}_{\mathit{fsym},\epsilon}(t:\mathit{term}[@V,@F]) : \mathit{bool} <=$
$\texttt{assume}\ ?\mathit{apply}(t);\ \neg \mathit{forall}.\mathit{list}(?\mathit{var}, \mathit{args}(t))$
$\texttt{procedure}\ \Delta^{1,\mathbf{1}}_{\mathit{args},\mathbf{11}}(t:\mathit{term}[@V,@F]) : \mathit{bool} <= \texttt{assume}\ ?\mathit{apply}(t);\ \mathit{false}$
$\texttt{procedure}\ \Delta^{1,\mathbf{2}}_{\mathit{args},\mathbf{12}}(t:\mathit{term}[@V,@F]) : \mathit{bool} <= \texttt{assume}\ ?\mathit{apply}(t);\ \mathit{true}$

Figure 4.3: Automatically synthesized difference procedures for selectors

Theorem 4.23. *Let $\mathit{sel}_j : \tau \to \tau_j$ be a (h,ϱ)-argument-bounded selector for $\tau = \mathit{str}[@A_1,\ldots,@A_k]$, $h \in \{1,\ldots,k\}$, and $\varrho \in \mathit{Pos}(\tau_j)$. Then a (h,ϱ)-difference procedure for sel_j is given by*

$$\texttt{procedure}\ \Delta^{1,h}_{\mathit{sel}_j,\varrho}(x:\tau) : \mathit{bool} <= \quad \texttt{assume}\ ?\mathit{cons}(x);\ \ \bigvee_{(j',\pi')\in\Pi} \#^{\geqslant 1}_{\tau_{j'}}\big(\mathit{sel}_{j'}(x),\pi'\big)$$

for $\Pi := \mathit{Occ}_{@A_h}(\mathit{cons}) \setminus \{(j,\varrho)\}$ and the str-constructor cons that sel_j belongs to.

Proof. Requirement (1) of Definition 4.21 is trivially satisfied. To show that requirement (2) is satisfied as well, let $\mathit{cons} : \tau_1 \times \ldots \times \tau_n \to \tau$ be the *str*-constructor that sel_j belongs to and let $q \in \mathbb{V}(P)_{\theta(\tau)}$ such that $\mathit{eval}_P(c_{\mathit{sel}_j}[q]) = \mathit{true}$. Thus $q = \mathit{cons}(q_1,\ldots,q_n)$ for some $q_i \in \mathbb{V}(P)_{\theta(\tau_i)}$.

Since $\tau_j\|_\varrho = \tau\|_h = @A_h$, we have (†) $(j,\varrho) \in \mathit{Occ}_{@A_h}(\mathit{cons})$. Hence:

$$\begin{aligned}
& \#_{\theta(\tau)}(q,h) > \#_{\theta(\tau_j)}(\mathit{eval}_P(\mathit{sel}_j(q)),\varrho) \\
\iff\ & \textstyle\sum_{(j',\pi')\in \mathit{Occ}_{@A_h}(\mathit{cons})} \#_{\theta(\tau_{j'})}(q_{j'},\pi') > \#_{\theta(\tau_j)}(q_j,\varrho) && \text{; def. of } \# \\
\iff\ & \textstyle\sum_{(j',\pi')\in \mathit{Occ}_{@A_h}(\mathit{cons})\setminus\{(j,\varrho)\}} \#_{\theta(\tau_{j'})}(q_{j'},\pi') > 0 && \text{; by (†)} \\
\iff\ & \#_{\theta(\tau_{j'})}(q_{j'},\pi') \geq 1 \text{ for some } (j',\pi') \in \Pi && \text{; def. of } \Pi \\
\iff\ & \mathit{eval}_P\big(\#^{\geqslant 1}_{\tau_{j'}}(q_{j'},\pi')\big) = \mathit{true} \text{ for some } (j',\pi') \in \Pi && \text{; Lemma 4.13} \\
\iff\ & \mathit{eval}_P\big(\Delta^{1,h}_{\mathit{sel}_j,\varrho}(q)\big) = \mathit{true}
\end{aligned}$$

Thus $\Delta^{1,h}_{\mathit{sel}_j,\varrho}$ is indeed a (h,ϱ)-difference procedure for sel_j. □

Example 4.24. Figure 4.3(b) shows the difference procedures for the (π, ϱ)-argument-bounded selectors from Example 4.20 with $\pi \neq \epsilon$.

The body of $\Delta^{1,\mathbf{1}}_{hd,\epsilon}$ uses predicate $\#^{\geqslant 1}_{list[@A]}(k, \mathbf{1})$ discussed in Example 4.14 (p. 93). Procedure $\Delta^{1,\mathbf{2}}_{fsym,\epsilon}$ decides if $args(t)$ contains a function symbol $f : @F$ and uses predicate $\#^{\geqslant 1}_{list[term[@V,@F]]}(args(t), \mathbf{12})$ from Example 4.15 (p. 94). ♢

The automated synthesis of difference procedures for argument-bounded *procedures* (e. g., *last* and *split*) is described in Section 4.3.2.

4.2.4 Argument-Bounded Functions of Arbitrary Arity

The notion of argument-boundedness and difference procedures can be generalized in a straightforward way to functions of arbitrary arity:

Definition 4.25 (Argument-bounded functions)**.** *A function*

$$f : \tau_1 \times \ldots \times \tau_n \to \tau'$$

with context requirement c_f is (p, π, ϱ)-argument-bounded *for $p \in \{1, \ldots, n\}$ and type positions $\pi \in Pos(\tau_p)$ and $\varrho \in Pos(\tau')$ iff*

1. *τ_p is a base type with $\tau_p\|_\pi = \tau'\|_\varrho$ and*
2. *for all grounding type substitutions $\theta \in GndSubst_{\Omega(P)}(\tau_1, \ldots, \tau_n)$ and all values $q_1, \ldots, q_n$ with $q_i \in \mathbb{V}(P)_{\theta(\tau_i)}$ for $i = 1, \ldots, n$ such that $eval_P(c_f[q_1, \ldots, q_n]) = true$:*

$$\#_{\theta(\tau_p)}(q_p, \pi) \geq \#_{\theta(\tau')}(eval_P(f(q_1, \ldots, q_n)), \varrho)\,.$$

Example 4.26. Procedure *filter* (cf. Figure 1.3 on p. 6) is $(2, \epsilon, \epsilon)$-argument-bounded, because the list of all elements x in k that satisfy $p(x)$ is not longer than k. ♢

Definition 4.27 (Difference function)**.** *For a (p, π, ϱ)-argument-bounded function $f : \tau_1 \times \ldots \times \tau_n \to \tau'$ with context requirement c_f, $\Delta^{p,\pi}_{f,\varrho} : \tau_1 \times \ldots \times \tau_n \to bool$ is a* (p, π, ϱ)-difference function for *f iff $\Delta^{p,\pi}_{f,\varrho}$ also has context requirement c_f and for all type substitutions $\theta \in GndSubst_{\Omega(P)}(\tau_1, \ldots, \tau_n)$ and all values $q_1, \ldots, q_n$ with $q_i \in \mathbb{V}(P)_{\theta(\tau_i)}$ for $i = 1, \ldots, n$ such that $eval_P(c_f[q_1, \ldots, q_n]) = true$:*

1. *$eval_P\big(\Delta^{p,\pi}_{f,\varrho}(q_1, \ldots, q_n)\big) \in \{true, false\}$ and*
2. *$eval_P\big(\Delta^{p,\pi}_{f,\varrho}(q_1, \ldots, q_n)\big) = true \iff$*
   $\#_{\theta(\tau_p)}(q_p, \pi) > \#_{\theta(\tau')}(eval_P(f(q_1, \ldots, q_n)), \varrho)\,.$

Example 4.28. The $(2, \epsilon, \epsilon)$-difference function $\Delta^{2,\epsilon}_{filter,\epsilon} : (@A \to bool) \times list[@A] \to bool$ needs to return *true* iff at least one element of the list does *not* satisfy the given predicate, because *filter* will return a proper sublist of the input list in this case. ♢

4.2.5 Discussion of Different Size Measures

One naturally wonders if there might be simpler alternatives to the size measure from Definition 4.7 (p. 91). Let us discuss some alternatives:

Counting only reflexive constructors. The size measure in [86, 96] counts only reflexive constructors (with weight 1). Such a size measure is inappropriate for our setting:

If we assumed that $\#(\mathsf{0}, \epsilon)$ was 0, then $\#(\mathsf{0} :: \varepsilon, \mathbf{1})$ would also be 0 (instead of 1). Thus $\#(\mathsf{0} :: \varepsilon, \mathbf{1}) = \#(\varepsilon, \mathbf{1})$, which makes a generic implementation of a difference function like $\Delta^{1,\mathbf{1}}_{hd,\epsilon} : \mathit{list}[@A] \to \mathit{bool}$ impossible. Since we do not know how type variable $@A$ will be instantiated in subsequent calls, we cannot generically find out if the list contains at least one element of non-zero size.

By ensuring that $\#(q, \epsilon) \neq 0$ for all values $q \in \mathcal{T}(\Sigma(P)^{\mathsf{c}})$, we have $\#(k, \mathbf{1}) = 0$ iff $k = \varepsilon$, which facilitates the implementation of difference functions such as $\Delta^{1,\mathbf{1}}_{hd,\epsilon}$ and $\Delta^{1,\mathbf{2}}_{\mathit{fsym},\epsilon}$ that do not exist in [86, 96].

Counting *all* constructors. Example 4.18 (p. 95) shows that a naïve size measure $\overline{\#} : \mathcal{T}(\Sigma^{\mathsf{c}}) \to \mathbb{N}$ that simply counts *all* constructors in a term $t \in \mathcal{T}(\Sigma(P)^{\mathsf{c}})$, regardless of the type constructor they belong to, is inappropriate for automated termination analysis: Since

$$\overline{\#}(\varepsilon) = 1 \not\geq 3 = \overline{\#}\big((\varepsilon \bullet \varepsilon)\big) = \overline{\#}\big(\mathit{eval}_P(\mathit{split}(\varepsilon))\big)\,,$$

split would not be argument-bounded wrt. $\overline{\#}$, so Mergesort could not be proved terminating as in Example 4.46 (p. 116).

Counting all *str*-constructors for some type constructor *str*. Parameterizing $\overline{\#}$ with a type constructor *str* would not solve the problem, because $(\varepsilon \bullet \varepsilon)$ contains *list*-constructor ε twice, so again

$$\overline{\#}(\varepsilon, \mathit{list}) = 1 \not\geq 2 = \overline{\#}\big((\varepsilon \bullet \varepsilon), \mathit{list}\big) = \overline{\#}\big(\mathit{eval}_P(\mathit{split}(\varepsilon)), \mathit{list}\big)\,.$$

Parameterizing # with a *type component* instead of a *type symbol*. There are two reasons against this option: Firstly, we would encounter the same problem as in the previous option:

$$\overline{\#}(\varepsilon, \mathit{list}[\mathbb{N}]) = 1 \not\geq 2 = \overline{\#}\big((\varepsilon \bullet \varepsilon), \mathit{list}[\mathbb{N}]\big) = \overline{\#}\big(\mathit{eval}_P(\mathit{split}(\varepsilon)), \mathit{list}[\mathbb{N}]\big)$$

Secondly, specifying a *type component* instead of a *type position* (that addresses a type symbol) would put unnecessary constraints on the definition of argument-boundedness. For example, our definition of the size measure considers procedure *map* (cf. Figure 1.3 on p. 6) as $(2, \epsilon, \epsilon)$-argument-bounded (as the result list is just as long as the input list). However, the list types are

different: $list[@A] \neq list[@B]$. Considering type position ϵ as a reference to type symbol *list* instead of the whole type component makes our approach more widely applicable.

4.3 Estimation Proofs

To show that a procedure computes an argument-bounded function, we need to show an inequality of the form $\#(t_1, \pi_1) \geq \#(t_2, \pi_2)$. A proof of such an inequality is called an *estimation proof*, which can be obtained using the estimation calculus of Definition 4.29 below. It extends the calculus from [86, 96] by type positions π and rule *(7)*, cf. Example 4.31 (p. 102).

In addition, the estimation calculus is used to synthesize difference procedures for argument-bounded procedures and to generate termination hypotheses for recursively defined procedures.

4.3.1 Estimation Calculus

The estimation calculus is used to prove inequalities $\#(t_1, \pi_1) \geq \#(t_2, \pi_2)$. The inequalities to be shown are given by some set E. When proving an inequality, a clause Δ (called a *difference equivalent of* E) is synthesized such that at least one of the proved inequalities is strict iff one of the literals in Δ holds. Furthermore, a clause ∇ (called *determination clause*) is synthesized that contains those literals that need to hold in order for the estimation proof to be valid.

Definition 4.29. *For a terminating program P, let $\Gamma^{p,\pi}_{\varrho}$ be a family of (p, π, ϱ)-argument-bounded function symbols in P. Given a* call context *$C \in \mathcal{CL}(\Sigma, \mathcal{V})$, the* estimation calculus *is defined by:*

Language: *Estimation triples of the form $\langle \nabla, \Delta, E \rangle$ with $\nabla, \Delta \in \mathcal{CL}(\Sigma, \mathcal{V})$ and $E \subset_{fin} \mathcal{E}(\Sigma(P), \mathcal{V}) := \{(t_1, \pi_1) \succcurlyeq (t_2, \pi_2) \mid t_i \in \mathcal{T}(\Sigma(P), \mathcal{V})_{\tau_i}$ for some base types τ_1, τ_2, $\pi_i \in Pos(\tau_i)$ for $i = 1, 2$ and $\tau_1\|_{\pi_1} = \tau_2\|_{\pi_2}\}$.*

Inference Rules: *The estimation rules are shown in Figure 4.4. They are given for each type constructor str and all data constructors $cons \in \mathcal{C}_{str}$, $rcons \in \mathcal{C}^{\mathsf{refl}}_{str}$, and $ircons, ircons_1, ircons_2 \in \mathcal{C}^{\mathsf{irr}}_{str}$.*

In the definition of the estimation rules, we write $C \vdash ?cons(t)$ iff (i) $t = cons(\ldots)$ or (ii) $?cons(t) \in C$ or (iii) $\neg ?cons'(t) \in C$ for all $cons' \in \mathcal{C}_{str} \setminus \{cons\}$.

$SEL_j(t)$ stands for t_j if $t = cons(\ldots, t_j, \ldots)$, and abbreviates $sel_j(t)$ otherwise.

Symbol "$\uplus$" denotes the union of two disjoint sets.

Deduction: *We write* $\langle \nabla_0, \Delta_0, E_0\rangle \Rrightarrow_{\Gamma,C} \langle \nabla_1, \Delta_1, E_1\rangle$ *iff* $\langle \nabla_1, \Delta_1, E_1\rangle$ *results from* $\langle \nabla_0, \Delta_0, E_0\rangle$ *by applying some estimation rule.* $\Rrightarrow^*_{\Gamma,C}$ *denotes the reflexive and transitive closure of* $\Rrightarrow_{\Gamma,C}$. $\langle \nabla_0, \Delta_0, E_0\rangle \Rrightarrow^*_{\Gamma,C} \langle \nabla_n, \Delta_n, E_n\rangle$ *is called a* deduction *of* $\langle \nabla_n, \Delta_n, E_n\rangle$ *from* $\langle \nabla_0, \Delta_0, E_0\rangle$.

We use the notation $\vdash_{\Gamma,C} \langle \nabla, \Delta, (t_1, \pi_1) \succcurlyeq (t_2, \pi_2)\rangle$ *iff*

$$\langle \emptyset, \emptyset, \{(t_1, \pi_1) \succcurlyeq (t_2, \pi_2)\}\rangle \Rrightarrow^*_{\Gamma,C} \langle \nabla, \Delta, \emptyset\rangle\,.$$

$(t_1, \pi_1) \geqslant_{\Gamma,C} (t_2, \pi_2)$ *denotes the existence of a* determination clause ∇ *and a* difference equivalent Δ *with* $\vdash_{\Gamma,C} \langle \nabla, \Delta, (t_1, \pi_1) \succcurlyeq (t_2, \pi_2)\rangle$.

Example 4.30. We get the following estimation proof for call context $C := \{\neg\, ?\varepsilon(k)\}$:

$$\begin{aligned}
&\langle \emptyset, \emptyset,\ \{(k, \epsilon) \succcurlyeq (\mathit{filter}(g, \mathit{tl}(k)), \epsilon)\}\rangle \\
\Rrightarrow_{\Gamma,C}\ &\langle \{\mathit{true}\}, \{\Delta^{2,\epsilon}_{\mathit{filter},\epsilon}(g, \mathit{tl}(k))\},\ \{(k, \epsilon) \succcurlyeq (\mathit{tl}(k), \epsilon)\}\rangle && \text{; by (5)} \\
\Rrightarrow_{\Gamma,C}\ &\langle \{\mathit{true}\}, \{\mathit{true},\ \Delta^{2,\epsilon}_{\mathit{filter},\epsilon}(g, \mathit{tl}(k))\},\ \{(\mathit{tl}(k), \epsilon) \succcurlyeq (\mathit{tl}(k), \epsilon)\}\rangle && \text{; by (4)} \\
\Rrightarrow_{\Gamma,C}\ &\langle \{\mathit{true}\}, \{\mathit{true},\ \Delta^{2,\epsilon}_{\mathit{filter},\epsilon}(g, \mathit{tl}(k))\},\ \emptyset\rangle && \text{; by (1)}
\end{aligned}$$

Since the difference equivalent is a disjunction that contains literal *true*, we can simplify it to $\Delta = \{\mathit{true}\}$. ♢

Example 4.31. Estimation rule *(7) Constructor Wrapping* is useful when the right-hand side of an inequality contains a constructor such as "•" (cf. Figure 2.1 on p. 31) that just wraps the item of interest:

$$\begin{aligned}
&\langle \nabla, \Delta, E \uplus \{(t, \varrho) \succcurlyeq ((t_1 \bullet t_2), \mathbf{1})\}\rangle \\
\Rrightarrow_{\Gamma,C}\ &\langle \nabla, \Delta, \{(t, \varrho) \succcurlyeq (t_1, \epsilon)\}\rangle
\end{aligned}$$

Thus it suffices to show $\#(t, \varrho) \geq \#(t_1, \epsilon)$ in order to show to show $\#(t, \varrho) \geq \#((t_1 \bullet t_2), \mathbf{1})$. ♢

To formally state soundness of the estimation calculus, we use expressions of the form

1. $(t_1, \pi_1) \geqslant_{\#} (t_2, \pi_2)$ and
2. $(t_1, \pi_1) >_{\#} (t_2, \pi_2)$

for terms $t_i \in \mathcal{T}(\Sigma(P), \mathcal{V})_{\tau_i}$ and type positions $\pi_i \in \mathit{Pos}(\tau_i)$, $i = 1, 2$, with $\tau_1\|_{\pi_1} = \tau_2\|_{\pi_2}$. Such expressions are *true* iff

1. $\#(\mathit{eval}_P(t_1), \pi_1) \geq \#(\mathit{eval}_P(t_2), \pi_2)$ or
2. $\#(\mathit{eval}_P(t_1), \pi_1) > \#(\mathit{eval}_P(t_2), \pi_2)$, respectively.

For $e := (t_1, \pi_1) \succcurlyeq (t_2, \pi_2) \in \mathcal{E}(\Sigma(P), \mathcal{V})$, we define $e^{\geqslant}$ as $(t_1, \pi_1) \geqslant_{\#} (t_2, \pi_2)$ and $e^{>}$ as $(t_1, \pi_1) >_{\#} (t_2, \pi_2)$.

Identity

$$(1)\ \frac{\langle \nabla, \Delta, E \uplus \{(t,\pi) \succcurlyeq (t,\pi)\}\rangle}{\langle \nabla, \Delta, E\rangle}$$

Equivalence

$$(2)\ \frac{\langle \nabla, \Delta, E \uplus \{(t_1,\epsilon) \succcurlyeq (t_2,\epsilon)\}\rangle}{\langle \nabla, \Delta, E\rangle} \quad \text{, if } C \vdash ?ircons_1(t_1) \text{ and } C \vdash ?ircons_2(t_2)$$

Strong Estimation

$$(3)\ \frac{\langle \nabla, \Delta, E \uplus \{(t_1,\epsilon) \succcurlyeq (t_2,\epsilon)\}\rangle}{\langle \nabla, \Delta \cup \{true\}, E\rangle} \quad \text{, if } C \vdash ?rcons(t_1) \text{ and } C \vdash ?ircons(t_2)$$

Strong Embedding

$$(4)\ \frac{\langle \nabla, \Delta, E \uplus \{(t_1,\epsilon) \succcurlyeq (t_2,\pi_2)\}\rangle}{\langle \nabla, \Delta \cup \{true\}, E \cup \{(SEL_j(t_1),\pi_1) \succcurlyeq (t_2,\pi_2)\}\rangle} \quad ,$$
if $C \vdash ?rcons(t_1)$ and $(j,\pi_1) \in Occ_{str}(rcons)$

Argument Estimation

$$(5)\ \frac{\langle \nabla, \Delta, E \uplus \{(t',\pi') \succcurlyeq (f(t_1,\ldots,t_n),\varrho)\}\rangle}{\langle \nabla \cup \{c_f[t_1,\ldots,t_n]\}, \Delta \cup \{\Delta^{p,\pi}_{f,\varrho}(t_1,\ldots,t_n)\}, E \cup \{(t',\pi') \succcurlyeq (t_p,\pi)\}\rangle} \quad ,$$
if $f \in \Gamma^{p,\pi}_{\varrho}$

Weak Embedding

$$(6)\ \frac{\langle \nabla, \Delta, E \uplus \{(t_1,\epsilon) \succcurlyeq (t_2,\epsilon)\}\rangle}{\langle \nabla, \Delta, E \cup \{\,(SEL_j(t_1),\pi) \succcurlyeq (SEL_j(t_2),\pi) \mid (j,\pi) \in Occ_{str}(rcons)\}\rangle} \quad ,$$
if $C \vdash ?rcons(t_1)$ and $C \vdash ?rcons(t_2)$

Constructor Wrapping

$$(7)\ \frac{\langle \nabla, \Delta, E \uplus \{(t,\varrho) \succcurlyeq (cons(t_1,\ldots,t_n), h\pi')\}\rangle}{\langle \nabla, \Delta, E \cup \{(t,\varrho) \succcurlyeq (t_j,\pi\pi')\}\rangle} \quad ,$$
if $Occ_{@A_h}(cons) = \{(j,\pi)\}$

Minimum

$$(8)\ \frac{\langle \nabla, \Delta, E \uplus \{(t_1,\epsilon) \succcurlyeq (t_2,\epsilon)\}\rangle}{\langle \nabla, \Delta \cup \{\,?rcons(t_1) \mid rcons \in \mathcal{C}^{\mathsf{refl}}_{str}\}, E\rangle} \quad \text{, if } C \vdash ?ircons(t_2)$$

Strong Direct Estimation

$$(9)\ \frac{\langle \nabla, \Delta, E \uplus \{(t_1,\epsilon) \succcurlyeq (t_2,\epsilon)\}\rangle}{\langle \nabla, \Delta \cup \{true\}, E\rangle} \quad \text{, if } t_1 > t_2 \in C$$

Weak Direct Estimation

$$(10)\ \frac{\langle \nabla, \Delta, E \uplus \{(t_1,\epsilon) \succcurlyeq (t_2,\epsilon)\}\rangle}{\langle \nabla, \Delta \cup \{t_1 \neq t_2\}, E\rangle} \quad \text{, if } \neg\, t_2 > t_1 \in C$$

Figure 4.4: Inference rules of the estimation calculus

Lemma 4.32. *The estimation rules are* sound*: For each*

$$\langle \nabla, \Delta, E \uplus \{e\} \rangle \Rrightarrow_{\Gamma,C} \langle \nabla \cup \nabla', \Delta \cup \Delta', E \cup E' \rangle$$

the following formulas are true (where $x_1, \ldots, x_k$ are all variables in C, ∇, Δ, E, and e such that $x_i : \tau_i$ for $i = 1, \ldots, k$):

1. $\forall x_1 : \tau_1, \ldots, x_k : \tau_k.\ \bigwedge C \wedge \bigwedge(\nabla \cup \nabla') \wedge \bigwedge E'^{\geqslant} \rightarrow e^{\geqslant}$
2. $\forall x_1 : \tau_1, \ldots, x_k : \tau_k.\ \bigwedge C \wedge \bigwedge(\nabla \cup \nabla') \wedge \bigwedge E'^{\geqslant} \rightarrow \left[\bigvee(\Delta' \cup E'^{>}) \leftrightarrow e^{>}\right]$

Proof. The deduction step $\Rrightarrow_{\Gamma,C}$ uses an estimation rule, so we prove the lemma by case analysis over the estimation rule that is applied.

To increase readability, we just write t, C, $E'^{\geqslant}$, ... to represent the corresponding ground instances $\sigma(t)$, $\sigma(C)$, $\sigma(E'^{\geqslant})$, ... for a terminating program $P' \supseteq P$, arbitrary $q_1, \ldots, q_k \in \mathbb{V}(P')$, and $\sigma := \{x_1/q_1, \ldots, x_k/q_k\}$. This abbreviation is "sound", because $\langle \nabla_0, \Delta_0, E_0 \rangle \Rrightarrow_{\Gamma,C} \langle \nabla_1, \Delta_1, E_1 \rangle$ entails that $\langle \sigma(\nabla_0), \sigma(\Delta_0), \sigma(E_0) \rangle \Rrightarrow_{\Gamma,C} \langle \sigma(\nabla_1), \sigma(\Delta_1), \sigma(E_1) \rangle$ by the definition of the estimation rules.

(1) *Identity: 1.* $\#(eval_P(t), \pi) \geq \#(eval_P(t), \pi)$. *2.* $\Delta' = E' = \emptyset$, so $\bigvee(\Delta' \cup E'^{>})$ is *false* and indeed $\#(eval_P(t), \pi) \not> \#(eval_P(t), \pi)$.

(2) *Equivalence: 1.* $\bigwedge C$ entails that both $eval_P(t_1)$ and $eval_P(t_2)$ are of the form $ircons(\ldots)$ for an irreflexive constructor *ircons*. Thus $\#(eval_P(t_1), \epsilon) = 1 \geq 1 = \#(eval_P(t_2), \epsilon)$. *2.* $\Delta' = E' = \emptyset$, so $\bigvee(\Delta' \cup E'^{>})$ is *false* and indeed $\#(eval_P(t_1), \epsilon) = 1 \not> 1 = \#(eval_P(t_2), \epsilon)$.

(3) *Strong Estimation: 1.* $\bigwedge C$ entails that $eval_P(t_1) = rcons(\ldots)$ and that $eval_P(t_2) = ircons(\ldots)$. Thus $\#(eval_P(t_1), \epsilon) = 2 + \ldots \geq 1 = \#(eval_P(t_2), \epsilon)$, where "..." is some natural number ≥ 0. *2.* $\Delta' = \{true\}$, so $\bigvee(\Delta' \cup E'^{>})$ is *true* and indeed $\#(eval_P(t_1), \epsilon) = 2 + \ldots > 1 = \#(eval_P(t_2), \epsilon)$.

(4) *Strong Embedding: 1.* $\bigwedge C$ entails that $eval_P(t_1) = rcons(t'_1, \ldots, t'_n)$ for some $t'_1, \ldots, t'_n$. Thus we conclude from $\bigwedge E'^{\geqslant}$ that $\#(t'_j, \pi_1) = \#(eval_P(SEL_j(t_1)), \pi_1) \geq \#(eval_P(t_2), \pi_2)$, so $\#(eval_P(t_1), \epsilon) = 2 + \#(t'_j, \pi_1) + \ldots \geq \#(eval_P(t_2), \pi_2)$. *2.* $\Delta' = \{true\}$, so $\bigvee(\Delta' \cup E'^{>})$ is *true* and indeed we have $\#(eval_P(t_1), \epsilon) = 2 + \#(t'_j, \pi_1) + \ldots \geq 2 + \#(eval_P(t_2), \pi_2) > \#(eval_P(t_2), \pi_2)$.

(5) *Argument Estimation: 1.* $\bigwedge \nabla'$ entails $eval_P(c_f[t_1, \ldots, t_n]) = true$. Using (i) $\bigwedge E'^{\geqslant}$ and (ii) $f \in \Gamma_\varrho^{p,\pi}$, we get

$$\#(eval_P(t'), \pi') \overset{\text{(i)}}{\geq} \#(eval_P(t_p), \pi) \overset{\text{(ii)}}{\geq} \#(eval_P(f(t_1, \ldots, t_n)), \varrho)\ .$$

2. If $\bigvee \Delta'$ is true, then inequality (ii) is strict. If $\bigvee E'^{>}$ is true, then inequality (i) is strict. If neither $\bigvee \Delta'$ nor $\bigvee E'^{>}$, then both (i) and (ii) are equalities, so $\#(eval_P(t'), \pi') \not> \#(eval_P(f(t_1, \ldots, t_n)), \varrho)$.

(6) *Weak Embedding: 1.* $\bigwedge C$ entails that $eval_P(t_1) = rcons(t'_1, \ldots, t'_n)$ and $eval_P(t_2) = rcons(t''_1, \ldots, t''_n)$ for some $t'_1, \ldots, t'_n, t''_1, \ldots, t''_n$. So $\bigwedge E'^{\geqslant}$ entails

$$\begin{aligned} & \#(t'_j, \pi) = \#(eval_P(SEL_j(t_1)), \pi) \\ (\dagger) \qquad & \geq \#(eval_P(SEL_j(t_2)), \pi) \\ & = \#(t''_j, \pi) \end{aligned}$$

for all $(j, \pi) \in Occ_{str}(rcons)$. Hence

$$\begin{aligned} \#(eval_P(t_1), \epsilon) &= 2 + \textstyle\sum_{(j,\pi)\in Occ_{str}(rcons)} \#(t'_j, \pi) \\ &\geq 2 + \textstyle\sum_{(j,\pi)\in Occ_{str}(rcons)} \#(t''_j, \pi) \qquad ; \text{by } (\dagger) \\ &= \#(eval_P(t_2), \epsilon)\,. \end{aligned}$$

2. If $\bigvee E'^{>}$, then one of the inequalities (†) is strict, which entails $\#(eval_P(t_1), \epsilon) > \#(eval_P(t_2), \epsilon)$. Otherwise (†) is an equality for each (j, π), so $\#(eval_P(t_1), \epsilon) \not> \#(eval_P(t_2), \epsilon)$.

(7) *Constructor Wrapping: 1.* $eval_P(cons(t_1, \ldots, t_n)) = cons(t'_1, \ldots, t'_n)$ for $t'_i := eval_P(t_i)$ and thus:

$$\begin{aligned} \#(eval_P(t), \varrho) &\overset{(\dagger)}{\geq} \#(t'_j, \pi\pi') && ; \text{by } \textstyle\bigwedge E'^{\geqslant} \\ &= \textstyle\sum_{(j',\pi'')\in Occ_{@A_h}(cons)} \#(t'_{j'}, \pi''\pi') && ; \text{def. of } (j, \pi) \\ &= \#(eval_P(cons(t_1, \ldots, t_n)), h\pi') && ; \text{def. of } \#\,. \end{aligned}$$

2. If $\bigvee E'^{>}$, inequality (†) above is strict, and hence $\#(eval_P(t), \varrho) > \#(eval_P(cons(t_1, \ldots, t_n)), \pi)$. Otherwise (†) is an equality, so in this case $\#(eval_P(t), \varrho) \not> \#(eval_P(cons(t_1, \ldots, t_n)), \pi)$.

(8) *Minimum: 1.* We have either $\#(eval_P(t_1), \epsilon) = 1$ or $\#(eval_P(t_1), \epsilon) = 2 + \ldots \geq 1$. $\bigwedge C$ entails that $eval_P(t_2) = ircons(\ldots)$, so in both cases $\#(eval_P(t_1), \epsilon) \geq 1 = \#(eval_P(t_2), \epsilon)$. *2.* If $\bigvee \Delta'$, then $eval_P(t_1) = rcons(\ldots)$ for some $rcons \in \mathcal{C}^{\mathsf{refl}}_{str}$, so $\#(eval_P(t_1), \epsilon) \geq 2 > 1 = \#(eval_P(t_2), \epsilon)$. Otherwise $eval_P(t_1)$ begins with an irreflexive constructor, so $\#(eval_P(t_1), \epsilon) = 1 \not> 1 = \#(eval_P(t_2), \epsilon)$.

(9) *Strong Direct Estimation: 1.* Both $eval_P(t_1)$ and $eval_P(t_2)$ are of the form $^+(\ldots{}^+(\mathtt{0})\ldots)$, i. e., they contain i_1 (or i_2, respectively) reflexive constructors $^+(\ldots)$ and one irreflexive constructor $\mathtt{0}$. $\bigwedge C$ entails $i_1 > i_2$, so $\#(eval_P(t_1), \epsilon) = 2i_1 + 1 > 2i_2 + 1 = \#(eval_P(t_2), \epsilon)$. *2.* $\Delta' = \{true\}$, so $\bigvee(\Delta' \cup E'^{>})$ is *true* and indeed the above inequality is strict.

(10) *Weak Direct Estimation: 1.* Again, $eval_P(t_1)$ and $eval_P(t_2)$ contain i_1 (or i_2, respectively) reflexive constructors $^+(\ldots)$ and one irreflexive

constructor 0. $\bigwedge C$ entails $i_1 \geq i_2$, so $\#(eval_P(t_1), \epsilon) = 2i_1 + 1 \geq 2i_2 + 1 = \#(eval_P(t_2), \epsilon)$. *2.* If $\bigvee(\Delta' \cup E'^{>}) = t_1 \not\approx t_2$ is true, then $i_1 \neq i_2$, so the above inequality is strict. Otherwise $i_1 = i_2$, and hence $\#(eval_P(t_1), \epsilon) = 2i_1 + 1 \not> 2i_2 + 1 = \#(eval_P(t_2), \epsilon)$. □

Remark 4.33. The estimation rules *(3) Strong Estimation* and *(8) Minimum* would be incorrect if the size measure assigned weight 1 to a reflexive constructor instead of 2 as in Definition 4.7 (p. 91):

- $\langle \emptyset, \emptyset, \{(apply(\mathsf{f}, \varepsilon), \epsilon) \succcurlyeq (var(\mathsf{v}), \epsilon)\}\rangle \Rrightarrow_{\Gamma,C} \langle \emptyset, \{true\}, \emptyset\rangle$ by *(3)*, and indeed $\#(apply(\mathsf{f}, \varepsilon), \epsilon) = 2 > 1 = \#(var(\mathsf{v}), \epsilon)$. In contrast, we would have $\#'(apply(\mathsf{f}, \varepsilon), \epsilon) = 1 \not> 1 = \#'(var(\mathsf{v}), \epsilon)$ for a size measure $\#'$ that counts *apply* as 1.
- $\langle \emptyset, \emptyset, \{(t, \epsilon) \succcurlyeq (var(\mathsf{v}), \epsilon)\}\rangle \Rrightarrow_{\Gamma,C} \langle \emptyset, \{?apply(t)\}, \emptyset\rangle$ by *(8)*. To ensure that $?apply(t)$ entails $\#(t, \epsilon) > 1 = \#(var(\mathsf{v}), \epsilon)$, the weight of the reflexive constructor *apply* needs to be greater than the weight of the irreflexive constructor *var*.

Instead of assigning weight 2 to a reflexive constructor, one could modify these two estimation rules, see [12]. Altogether, the estimation calculus would produce more complicated results (e. g., Δ would no longer be a disjunction of literals) without being able to prove more.

Therefore we prefer a slightly more complex size measure that distinguishes between reflexive and irreflexive data constructors. The size measure only exists on the meta-level (i. e., on a paper) but not in the implementation (i. e., in the estimation calculus). Thus the complexity of the size measure does not influence the performance of our method in practice, whereas a more complex estimation calculus would produce more complex results.

Lemma 4.34. *If* $\langle \nabla_0, \Delta_0, E_0\rangle \Rrightarrow^n_{\Gamma,C} \langle \nabla_n, \Delta_n, E_n\rangle$ *for some* $n \in \mathbb{N}$ *such that* $\nabla_0 = \Delta_0 = \emptyset$ *and* $E_0 = \{e_0\}$ *for some* $e_0 \in \mathcal{E}(\Sigma(P), \mathcal{V})$*, then the following formulas are true (where* $x_1, \ldots, x_k$ *are all variables in* C *and* e_0 *such that* $x_i : \tau_i$ *for all* $i = 1, \ldots, k$*):*

1. $\forall x_1 : \tau_1, \ldots, x_k : \tau_k.\ \bigwedge C \wedge \bigwedge \nabla_n \wedge \bigwedge E_n^{\geqslant} \to e_0^{\geqslant}$
2. $\forall x_1 : \tau_1, \ldots, x_k : \tau_k.\ \bigwedge C \wedge \bigwedge \nabla_n \wedge \bigwedge E_n^{\geqslant} \to \left[\bigvee(\Delta_n \cup E_n^{>}) \leftrightarrow e_0^{>}\right]$

Proof. We prove the statements by induction on the length n of the deduction.

$n = 0$: The tautologies

1. $\forall \ldots\ \bigwedge C \wedge e_0^{\geqslant} \to e_0^{\geqslant}$ and
2. $\forall \ldots\ \bigwedge C \wedge e_0^{\geqslant} \to \left[e_0^{>} \leftrightarrow e_0^{>}\right]$

are trivially true.

$n \rightsquigarrow n+1$: For a deduction

$$\langle \nabla_0, \Delta_0, E_0 \rangle \Rrightarrow^n_{\Gamma,C} \langle \nabla_n, \Delta_n, E_n \rangle \Rrightarrow_{\Gamma,C} \langle \nabla_{n+1}, \Delta_{n+1}, E_{n+1} \rangle$$

we have $E_{n+1} = (E_n \setminus \{e_n\}) \cup E'_{n+1}$ for some $e_n \in E_n$, and Lemma 4.32 yields:

$$\forall \ldots \bigwedge C \wedge \bigwedge \nabla_{n+1} \wedge \bigwedge E'^{\geqslant}_{n+1} \to e^{\geqslant}_n \tag{4.1}$$

$$\forall \ldots \bigwedge C \wedge \bigwedge \nabla_{n+1} \wedge \bigwedge E'^{\geqslant}_{n+1} \to \left[\bigvee\left((\Delta_{n+1} \setminus \Delta_n) \cup E'^{>}_{n+1}\right) \leftrightarrow e^{>}_n\right] \tag{4.2}$$

Since $E_{n+1} \supseteq E'_{n+1}$, we have

$$\forall \ldots \bigwedge C \wedge \bigwedge \nabla_{n+1} \wedge \bigwedge E^{\geqslant}_{n+1} \to \bigwedge E^{\geqslant}_n \tag{4.3}$$

due to (4.1) and $E_{n+1} \cup \{e_n\} \supseteq E_n$.

1. $\forall \ldots \bigwedge C \wedge \bigwedge \nabla_{n+1} \wedge \bigwedge E^{\geqslant}_{n+1} \to e^{\geqslant}_0$ is true, because $\nabla_{n+1} \supseteq \nabla_n$ and (4.3) allow us to use the induction hypothesis

   $$\forall \ldots \bigwedge C \wedge \bigwedge \nabla_n \wedge \bigwedge E^{\geqslant}_n \to e^{\geqslant}_0$$

   to show $e^{\geqslant}_0$ from $\bigwedge C \wedge \bigwedge \nabla_{n+1} \wedge \bigwedge E^{\geqslant}_{n+1}$.
2. $\forall \ldots \bigwedge C \wedge \bigwedge \nabla_{n+1} \wedge \bigwedge E^{\geqslant}_{n+1} \to \left[\bigvee(\Delta_{n+1} \cup E^{>}_{n+1}) \leftrightarrow e^{>}_0\right]$ is true: We assume $\bigwedge C \wedge \bigwedge \nabla_{n+1} \wedge \bigwedge E^{\geqslant}_{n+1}$. By $\nabla_{n+1} \supseteq \nabla_n$ and (4.3) we also have $\bigwedge \nabla_n$ and $\bigwedge E^{\geqslant}_n$. The induction hypothesis is

   $$\forall \ldots \bigwedge C \wedge \bigwedge \nabla_n \wedge \bigwedge E^{\geqslant}_n \to \left[\bigvee(\Delta_n \cup E^{>}_n) \leftrightarrow e^{>}_0\right]. \tag{4.4}$$

   "$\leftarrow$": If $e^{>}_0$ is true, then some $\delta \in \Delta_n \cup E^{>}_n$ is true by (4.4).

   Case $\delta \in \Delta_n$: Then also $\delta \in \Delta_{n+1}$.

   Case $\delta \in E^{>}_n$: If $\delta \neq e^{>}_n$, then $\delta \in E^{>}_{n+1}$. Otherwise $\delta = e^{>}_n$, so some $\delta' \in (\Delta_{n+1} \setminus \Delta_n) \cup E'^{>}_{n+1} \subseteq \Delta_{n+1} \cup E^{>}_{n+1}$ is true by (4.2).

   "$\rightarrow$": Assume that some $\delta \in \Delta_{n+1} \cup E^{>}_{n+1}$ is true.

   Case $\delta \in \Delta_{n+1}$: If $\delta \in \Delta_n$, then $e^{>}_0$ is true by (4.4). Otherwise $\delta \in \Delta_{n+1} \setminus \Delta_n$, so $e^{>}_n$ is true by (4.2). Thus $e^{>}_0$ is true by (4.4).

   Case $\delta \in E^{>}_{n+1}$: If $\delta \in (E_n \setminus \{e_n\})^{>}$, then $e^{>}_0$ is true by (4.4). If $\delta \in E'^{>}_{n+1}$, then $e^{>}_n$ is true by (4.2), so $e^{>}_0$ is true by (4.4). □

Theorem 4.35. *The estimation calculus is* sound*: If* $\vdash_{\Gamma,C} \langle \nabla, \Delta, (t_1, \pi_1) \succcurlyeq (t_2, \pi_2)\rangle$*, then the following formulas are true (where* $x_1, \ldots, x_n$ *are all variables in* C*,* t_1*, and* t_2 *such that* $x_i : \tau_i$ *for all* $i = 1, \ldots, n$*):*

1. $\forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ \bigwedge C \wedge \bigwedge \nabla \to (t_1, \pi_1) \geqslant_{\#} (t_2, \pi_2)$
2. $\forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ \bigwedge C \wedge \bigwedge \nabla \to \left[\bigvee \Delta \leftrightarrow (t_1, \pi_1) >_{\#} (t_2, \pi_2)\right]$

Proof. By Definition 4.29 (p. 101), $\vdash_{\Gamma,C} \langle \nabla, \Delta, (t_1, \pi_1) \succcurlyeq (t_2, \pi_2)\rangle$ means $\langle \emptyset, \emptyset, \{(t_1, \pi_1) \succcurlyeq (t_2, \pi_2)\}\rangle \Rrightarrow^*_{\Gamma,C} \langle \nabla, \Delta, \emptyset\rangle$. Thus the truth of the two formulas follows immediately from Lemma 4.34. □

Theorem 4.36. *The set*

$$\{\, (t_1, \pi_1) \succcurlyeq (t_2, \pi_2) \in \mathcal{E}(\Sigma(P), \mathcal{V}) \mid (t_1, \pi_1) \geqslant_{\Gamma,C} (t_2, \pi_2)\}$$

of provable size estimation problems is decidable.

Proof. Define $M := \{\, \langle \nabla, \Delta, E\rangle \mid \nabla, \Delta \in \mathcal{CL}(\Sigma, \mathcal{V}),\ E \subset_{\mathit{fin}} \mathcal{E}(\Sigma(P), \mathcal{V})\}$.

1. $(M, \Rrightarrow_{\Gamma,C})$ is a well-founded set: The idea is that after applying an estimation rule, the terms on the left-hand sides of the inequalities denote smaller *values* than before (all rules except (5) and (7)) or that the *terms* on the right-hand side get smaller (rules (5) and (7)).

   Formally, we assign a pair $\|E\| \in \mathbb{N} \times \mathbb{N}$ of natural numbers to component $E \subset_{\mathit{fin}} \mathcal{E}(\Sigma(P), \mathcal{V})$ of an estimation triple $\langle \nabla, \Delta, E\rangle$. Then we show that $\|E\|$ gets lexicographically smaller by each application of an estimation rule. Our definition of $\|E\|$ assumes an arbitrary, but fixed mapping $\sigma : \mathcal{V} \to \mathbb{V}(P)$ that assigns each variable $v \in \mathcal{V}$ a value $\sigma(v) \in \mathbb{V}(P)$. $\|E\|$ is defined by $\|E\| := (n_1, n_2)$, where

   $$n_1 := \sum_{(t_1,\pi_1) \succcurlyeq (t_2,\pi_2) \in E} \#(\mathit{eval}_P(\sigma(t_1)), \pi_1)$$
   $$n_2 := \sum_{(t_1,\pi_1) \succcurlyeq (t_2,\pi_2) \in E} |t_2|_{\mathcal{T}}$$

   for the number $|t_2|_{\mathcal{T}}$ of function and variable symbols in t_2.

   Now $\|E_0\| >_{\mathbb{N}\times\mathbb{N}} \|E_1\|$ for each $\langle \nabla_0, \Delta_0, E_0\rangle \Rrightarrow_{\Gamma,C} \langle \nabla_1, \Delta_1, E_1\rangle$ is obvious for rules (1), (2), (3), (8), (9), (10) of the estimation rules (cf. Figure 4.4 on p. 103), because $E_0 \supsetneqq E_1$ if one of these rules is applied.

   For (4), $\mathit{eval}_P(\sigma(t_1)) = \mathit{rcons}(t'_1, \ldots, t'_n)$ for some $t'_1, \ldots, t'_n$, so

   $$\begin{aligned}\#(\mathit{eval}_P(\sigma(t_1)), \epsilon) &= 2 + \#(t'_j, \pi_1) + \ldots \\ &> \#(t'_j, \pi_1) = \#(\mathit{eval}_P(\sigma(\mathit{SEL}_j(t_1))), \pi_1)\,.\end{aligned}$$

   For (5), $|f(t_1, \ldots, t_n)|_{\mathcal{T}} = 1 + |t_p|_{\mathcal{T}} + \ldots > |t_p|_{\mathcal{T}}$.

For (6), $eval_P(\sigma(t_1)) = rcons(t'_1, \ldots, t'_n)$ for some $t'_1, \ldots, t'_n$, so

$$\begin{aligned}\#(eval_P(\sigma(t_1)), \epsilon) &= 2 + \textstyle\sum_{(j,\pi) \in Occ_{str}(rcons)} \#(t'_j, \pi) \\ &> \textstyle\sum_{(j,\pi) \in Occ_{str}(rcons)} \#(t'_j, \pi) \\ &= \textstyle\sum_{(j,\pi) \in Occ_{str}(rcons)} \#(eval_P(\sigma(SEL_j(t_1))), \pi)\,.\end{aligned}$$

For (7), $|cons(t_1, \ldots, t_n)|_{\mathcal{T}} = 1 + |t_j|_{\mathcal{T}} + \ldots > |t_j|_{\mathcal{T}}$.

2. $\Rrightarrow_{\Gamma,C}$ is locally finite: Only finitely many instances of estimation rules are applicable to an estimation triple $\langle \nabla, \Delta, E \rangle$, because both E and the number of rules and data constructors are finite.

Thus a decision procedure for $(t_1, \pi_1) \geqslant_{\Gamma,C} (t_2, \pi_2)$ can just enumerate all $\langle \nabla, \Delta, E \rangle$ with $\langle \emptyset, \emptyset, \{(t_1, \pi_1) \succcurlyeq (t_2, \pi_2)\} \rangle \Rrightarrow_{\Gamma,C} \langle \nabla, \Delta, E \rangle$ such that no estimation rule is applicable to $\langle \nabla, \Delta, E \rangle$ and check if $E = \emptyset$. □

Similarly to [86, 96], a proof procedure for the estimation calculus in general needs to backtrack if a dead end is encountered in a deduction (a dead end is an estimation triple $\langle \nabla, \Delta, E \rangle$ with $E \neq \emptyset$ such that no estimation rule is applicable to $\langle \nabla, \Delta, E \rangle$). If more than one estimation rule is applicable to an estimation triple, then the next rule to be applied is chosen according to the following (decreasing) preference: *(1) Identity*, *(2) Equivalence*, *(3) Strong Estimation*, *(8) Minimum*, *(4) Strong Embedding*, *(5) Argument Estimation*, *(6) Weak Embedding*, *(7) Constructor Wrapping*, *(9) Strong Direct Estimation*, *(10) Weak Direct Estimation*. Obviously, backtracking to a choice point where an estimation rule has been applied that removes an inequality from E (i. e., estimation rules *(1)*, *(2)*, *(3)*, *(8)*, *(9)*, *(10)*) is not necessary. Estimation rules *(9)* and *(10)* are considered with lowest priority to avoid that literals like $t_1 > t_2$ or $\neg\, t_2 > t_1$ are unnecessarily marked as *used* if there is another possibility to prove the inequality; this is beneficial for the synthesis of optimized induction axioms, cf. Section 5.2.3.

Whenever a proof procedure for the estimation calculus finds a proof of $(t_1, \pi_1) \geqslant_{\Gamma,C} (t_2, \pi_2)$, we know that t_1 is at least as big as t_2 by Theorem 4.35. If no estimation proof exists, the inequality might still hold, because the estimation calculus is incomplete; for instance, just as its predecessor [86, 96] it cannot prove $(fac(n), \epsilon) \geqslant_{\Gamma,C} (n, \epsilon)$ for the factorial function *fac* and any $n \in \mathbb{V}(P)_{\mathbb{N}}$. However, it is powerful enough to solve termination problems that are relevant in practice, see Section 4.5.

4.3.2 Proving Argument-Boundedness of Procedures

Using the estimation calculus, we can prove argument-boundedness of a procedure f by analyzing the result terms of body B_f of procedure f. In the corresponding deductions, an additional rule may be used:

Definition 4.37 ($\vdash^f_{\Gamma,C}$)**.** *For a procedure*

$$\texttt{procedure}\ f(x_1:\tau_1,\ldots,x_n:\tau_n):\tau <= \texttt{assume}\ c_f;\ B_f$$

and some $p \in \{1,\ldots,n\}$, $\pi \in Pos(\tau_p)$, $\varrho \in Pos(\tau)$ such that τ_p is a base type with $\tau_p\|_\pi = \tau\|_\varrho$, $\vdash^f_{\Gamma,C}$ is defined as $\vdash_{\Gamma,C}$ (cf. Definition 4.29 on p. 101) except that the deduction may use the Argument Estimation *rule (5) for each direct recursive call $f(s_1,\ldots,s_n)$ in B_f:*

$$\frac{\langle \nabla, \Delta, E \uplus \{(t',\pi') \succcurlyeq (f(s_1,\ldots,s_n),\varrho)\}\rangle}{\langle \nabla \cup \{c_f[s_1,\ldots,s_n]\}, \Delta \cup \{\Delta^{p,\pi}_{f,\varrho}(s_1,\ldots,s_n)\}, E \cup \{(t',\pi') \succcurlyeq (s_p,\pi)\}\rangle}\ (5^f)$$

Theorem 4.38. *For a terminating and context correct procedure*

$$\texttt{procedure}\ f(x_1:\tau_1,\ldots,x_n:\tau_n):\tau <= \texttt{assume}\ c_f;\ B_f$$

without indirect recursive calls, let $\{\pi_1,\ldots,\pi_m\} := ResPos(B_f)$ be the result term positions of B_f.[7] *For each $i = 1,\ldots,m$, let $t_i := B_f|_{\pi_i}$ be the result term at position π_i and let $C_i := \{c_f\} \cup COND(B_f,\pi_i)$ be the call context of t_i. For some $p \in \{1,\ldots,n\}$, let $\pi \in Pos(\tau_p)$ and $\varrho \in Pos(\tau)$ be type positions such that*

1. *τ_p is a base type with $\tau_p\|_\pi = \tau\|_\varrho$ and*
2. *$\vdash^f_{\Gamma,C_i} \langle \nabla_i, \Delta_i, (x_p,\pi) \succcurlyeq (t_i,\varrho)\rangle$ for all $i = 1,\ldots,m$.*

Then f is (p,π,ϱ)-argument-bounded and

$$\texttt{procedure}\ \Delta^{p,\pi}_{f,\varrho}(x_1:\tau_1,\ldots,x_n:\tau_n):\mathit{bool} <= \texttt{assume}\ c_f;\ B_{\Delta_f}$$

is a (p,π,ϱ)-difference procedure for f, where B_{Δ_f} is derived from B_f by replacing each result term t_i with the disjunction $\bigvee \Delta_i$:

$$B_{\Delta_f} := B_f[\pi_1 \leftarrow \bigvee \Delta_1]\ldots[\pi_m \leftarrow \bigvee \Delta_m]$$

Proof. The idea is to prove the theorem by induction on the recursion structure of procedure f. In the base cases there are no recursive calls $f(s_1,\ldots,s_n)$, so $\vdash^f_{\Gamma,C_i} = \vdash_{\Gamma,C_i}$ and the theorem follows from the soundness of the estimation calculus (cf. Theorem 4.35 on p. 108). In the step cases the induction hypotheses allow us to assume that argument-boundedness holds for the direct recursive calls. Since this assumption is precisely what is formalized by rule (5^f), the soundness theorem of the estimation calculus extends to $\vdash^f_{\Gamma,C_i}$, which proves the theorem.

[7] We assume procedure bodies B_f as *let*-free in this chapter, so all *let*-expressions are implicitly eliminated as described on p. 41.

Formally, let $\theta \in \mathit{GndSubst}_{\Omega(P)}(\tau_1, \ldots, \tau_n)$. We prove by induction on $\succ_f^\theta$ that $\mathit{eval}_P(c_f[q_1, \ldots, q_n]) = \mathit{true}$ implies

(i) $\#(q_p, \pi) \geq \#(\mathit{eval}_P(f(q_1, \ldots, q_n)), \varrho)$ and

(ii) $\#(q_p, \pi) > \#(\mathit{eval}_P(f(q_1, \ldots, q_n)), \varrho) \iff$
$\mathit{eval}_P(\Delta_{f,\varrho}^{p,\pi}(q_1, \ldots, q_n)) = \mathit{true}$.

Base case: $(q_1, \ldots, q_n)$ is $\succ_f^\theta$-minimal. Let t_i be the corresponding result term; i. e., $\mathit{eval}_P(\sigma(c)) = \mathit{true}$ for all $c \in C_i$, $\sigma := \{x_1/q_1, \ldots, x_n/q_n\}$, and $\mathit{eval}_P(f(q_1, \ldots, q_n)) = \mathit{eval}_P(\sigma(t_i))$. Since $(q_1, \ldots, q_n)$ is $\succ_f^\theta$-minimal, there is no recursive call $f(s_1, \ldots, s_n)$ in t_i, so rule (5^f) cannot be used in $\vdash^f_{\Gamma,C_i} \langle \nabla_i, \Delta_i, (x_p, \pi) \succcurlyeq (t_i, \varrho)\rangle$. Thus $\vdash^f_{\Gamma,C_i} = \vdash_{\Gamma,C_i}$ and (i) follows from Theorem 4.35(1) and (ii) follows from Theorem 4.35(2) by construction of procedure $\Delta_{f,\varrho}^{p,\pi}$.

Step case: $(q_1, \ldots, q_n)$ is not $\succ_f^\theta$-minimal. Thus the corresponding result term t_i contains at least one recursive call $f(s_1, \ldots, s_n)$, which means that rule (5^f) can be used in $\vdash^f_{\Gamma,C_i} \langle \nabla_i, \Delta_i, (x_p, \pi) \succcurlyeq (t_i, \varrho)\rangle$. For each recursive call $f(s_1, \ldots, s_n)$ in t_i we have $(q_1, \ldots, q_n) \succ_f^\theta (q'_1, \ldots, q'_n)$, where $q'_j := \mathit{eval}_P(\sigma(s_j))$ for $j = 1, \ldots, n$. The induction hypotheses

(i′) $\#(q'_p, \pi) \geq \#(\mathit{eval}_P(f(q'_1, \ldots, q'_n)), \varrho)$ and

(ii′) $\#(q'_p, \pi) > \#(\mathit{eval}_P(f(q'_1, \ldots, q'_n)), \varrho) \iff$
$\mathit{eval}_P(\Delta_{f,\varrho}^{p,\pi}(q'_1, \ldots, q'_n)) = \mathit{true}$

guarantee that the use of rule (5^f) in $\vdash^f_{\Gamma,C_i} \langle \nabla_i, \Delta_i, (x_p, \pi) \succcurlyeq (t_i, \varrho)\rangle$ is sound in the sense of Lemma 4.32 (p. 104). Hence Theorem 4.35 again entails (i) and (ii).

Finally, by construction of $\Delta_{f,\varrho}^{p,\pi}$, this difference procedure has the same recursion structure as procedure f. Since procedure f terminates, so does procedure $\Delta_{f,\varrho}^{p,\pi}$. □

Example 4.39. Procedure *last* (cf. Figure 4.2 on p. 95) is $(1, \mathbf{1}, \epsilon)$-argument-bounded: There are two result terms $t_1 := \mathit{hd}(k)$ and $t_2 := \mathit{last}(\mathit{tl}(k))$ with call contexts $C_1 := \{?::(k), ?\varepsilon(\mathit{tl}(k))\}$ and $C_2 := \{?::(k), \neg\, ?\varepsilon(\mathit{tl}(k))\}$, respectively. The deductions in Figure 4.5 show that

- $\vdash^{\mathit{last}}_{\Gamma,C_1} \langle \nabla_1, \Delta_1, (k, \mathbf{1}) \succcurlyeq (\mathit{hd}(k), \epsilon)\rangle$ for $\Delta_1 := \{\Delta^{1,\mathbf{1}}_{\mathit{hd},\epsilon}(k)\}$ and
- $\vdash^{\mathit{last}}_{\Gamma,C_2} \langle \nabla_2, \Delta_2, (k, \mathbf{1}) \succcurlyeq (\mathit{last}(\mathit{tl}(k)), \epsilon)\rangle$ for
$\Delta_2 := \{\Delta^{1,\mathbf{1}}_{\mathit{tl},\mathbf{1}}(k),\ \Delta^{1,\mathbf{1}}_{\mathit{last},\epsilon}(\mathit{tl}(k))\}$.

$\bigvee \Delta_1$ simplifies to *false* and $\bigvee \Delta_2$ simplifies to *true* using the definition of the difference procedures (cf. Figure 4.3 on p. 98) and call contexts C_1 and C_2. Difference procedure $\Delta^{1,\mathbf{1}}_{\mathit{last},\epsilon}$ is shown in Figure 4.6: The last element of list k is smaller than the sum of the sizes of all list elements if the length of k is ≥ 2. ♢

$\langle \emptyset, \emptyset, \{(k, \mathbf{1}) \succcurlyeq (hd(k), \epsilon)\}\rangle$
$\Rrightarrow_{\Gamma, C_1} \langle \{?::(k)\}, \{\Delta^{1,\mathbf{1}}_{hd,\epsilon}(k)\}, \{(k, \mathbf{1}) \succcurlyeq (k, \mathbf{1})\}\rangle$; by (5)
$\Rrightarrow_{\Gamma, C_1} \langle \{?::(k)\}, \{\Delta^{1,\mathbf{1}}_{hd,\epsilon}(k)\}, \emptyset\rangle$; by (1)

$\langle \emptyset, \emptyset, \{(k, \mathbf{1}) \succcurlyeq (last(tl(k)), \epsilon)\}\rangle$
$\Rrightarrow_{\Gamma, C_2} \langle \{?::(tl(k))\}, \{\Delta^{1,\mathbf{1}}_{last,\epsilon}(tl(k))\}, \{(k, \mathbf{1}) \succcurlyeq (tl(k), \mathbf{1})\}\rangle$; by (5^{last})
$\Rrightarrow_{\Gamma, C_2} \langle \{?::(k), ?::(tl(k))\}, \{\Delta^{1,\mathbf{1}}_{tl,\mathbf{1}}(k), \Delta^{1,\mathbf{1}}_{last,\epsilon}(tl(k))\}, \{(k, \mathbf{1}) \succcurlyeq (k, \mathbf{1})\}\rangle$; by (5)
$\Rrightarrow_{\Gamma, C_2} \langle \{?::(k), ?::(tl(k))\}, \{\Delta^{1,\mathbf{1}}_{tl,\mathbf{1}}(k), \Delta^{1,\mathbf{1}}_{last,\epsilon}(tl(k))\}, \emptyset\rangle$; by (1)

Figure 4.5: Estimation proofs to show argument-boundedness of *last*

```
procedure Δ^{1,1}_{last,ε}(k : list[@A]) : bool <=
assume ?::(k);
if ?ε(tl(k))
  then false
  else true
end

procedure Δ^{2,ε}_{filter,ε}(p : @A → bool, k : list[@A]) : bool <=
if ?ε(k)
  then false
  else if p(hd(k))
         then Δ^{2,ε}_{filter,ε}(p, tl(k))
         else true
       end
end

procedure Δ^{1,ε}_{split,1}(k : list[@A]) : bool <=
if ?ε(k)
  then false
  else if ?ε(tl(k))
         then false
         else true
       end
end

procedure Δ^{1,ε}_{split,2}(k : list[@A]) : bool <=
if ?ε(k)
  then false
  else true
end
```

Figure 4.6: Difference procedures for argument-bounded procedures

Example 4.40. Procedure *filter* (cf. Figure 1.3 on p. 6) is $(2, \epsilon, \epsilon)$-argument-bounded: There are three result terms:

1. $t_1 := \varepsilon$ with call context $C_1 := \{?\varepsilon(k)\}$,
2. $t_2 := hd(k) :: \mathit{filter}(p, tl(k))$ with call context $C_2 := \{\neg\, ?\varepsilon(k), p(hd(k))\}$ and
3. $t_3 := \mathit{filter}(p, tl(k))$ with call context $C_3 := \{\neg\, ?\varepsilon(k),\ \neg\, p(hd(k))\}$.

From the estimation proofs

1. $\vdash^{\mathit{filter}}_{\Gamma, C_1} \langle \nabla_1, \Delta_1, (k, \epsilon) \succcurlyeq (\varepsilon, \epsilon)\rangle$ for $\Delta_1 := \emptyset$,
2. $\vdash^{\mathit{last}}_{\Gamma, C_2} \langle \nabla_2, \Delta_2, (k, \epsilon) \succcurlyeq (hd(k) :: \mathit{filter}(p, tl(k)), \epsilon)\rangle$ for $\Delta_2 := \{\Delta^{2,\epsilon}_{\mathit{filter},\epsilon}(p, tl(k))\}$, and
3. $\vdash^{\mathit{last}}_{\Gamma, C_3} \langle \nabla_3, \Delta_3, (k, \epsilon) \succcurlyeq (\mathit{filter}(p, tl(k)), \epsilon)\rangle$ for $\Delta_3 := \{\mathit{true}\}$

we get the difference procedure shown in Figure 4.6. This reflects the intuition that the returned sublist of k is shorter than k iff at least one element x of k does *not* satisfy $p(x)$. ♢

Example 4.41. Procedure *split* (cf. Figure 4.2 on p. 95) is $(1, \epsilon, \mathbf{1})$-argument-bounded, because:

1. $\vdash^{\mathit{split}}_{\Gamma, C_1} \langle \nabla_1, \Delta_1, (k, \epsilon) \succcurlyeq ((\varepsilon \bullet \varepsilon), \mathbf{1})\rangle$ for $C_1 := \{?\varepsilon(k)\}$ and $\Delta_1 := \emptyset$,
2. $\vdash^{\mathit{split}}_{\Gamma, C_2} \langle \nabla_2, \Delta_2, (k, \epsilon) \succcurlyeq ((k \bullet \varepsilon), \mathbf{1})\rangle$ for $C_2 := \{\neg\, ?\varepsilon(k),\ ?\varepsilon(tl(k))\}$ and $\Delta_2 := \emptyset$, and
3. $\vdash^{\mathit{split}}_{\Gamma, C_3} \langle \nabla_3, \Delta_3, (k, \epsilon) \succcurlyeq ((hd(k) :: \mathit{fst}(\mathit{split}(tl(tl(k)))) \bullet \ldots), \mathbf{1})\rangle$ for $C_3 := \{\neg\, ?\varepsilon(k),\ \neg\, ?\varepsilon(tl(k))\}$ and $\Delta_3 := \{\Delta^{1,\epsilon}_{tl,\epsilon}(tl(k)),\ \Delta^{1,\epsilon}_{\mathit{split},\mathbf{1}}(tl(tl(k))),\ \Delta^{1,\mathbf{1}}_{\mathit{fst},\epsilon}(\mathit{split}(tl(tl(k))))\}$.

Both $\bigvee \Delta_1$ and $\bigvee \Delta_2$ are *false*. $\bigvee \Delta_3$ can be simplified to *true*, because C_3 entails that $\Delta^{1,\epsilon}_{tl,\epsilon}(tl(k))$ evaluates to *true* (cf. Figure 4.3 on p. 98). The difference procedure $\Delta^{1,\epsilon}_{\mathit{split},\mathbf{1}}$ in Figure 4.6 reflects the intuition that the sublist $e_1 :: e_3 :: \ldots$ of $k = e_1 :: e_2 :: e_3 :: e_4 :: \ldots$ is strictly smaller than k iff k contains at least two elements.

Procedure *split* is also $(1, \epsilon, \mathbf{2})$-argument-bounded, because:

- $\vdash^{\mathit{split}}_{\Gamma, C_1} \langle \nabla_1, \Delta_1, (k, \epsilon) \succcurlyeq ((\varepsilon \bullet \varepsilon), \mathbf{2})\rangle$ for $C_1 := \{?\varepsilon(k)\}$ and $\Delta_1 := \emptyset$,
- $\vdash^{\mathit{split}}_{\Gamma, C_2} \langle \nabla_2, \Delta_2, (k, \epsilon) \succcurlyeq ((k \bullet \varepsilon), \mathbf{2})\rangle$ for $C_2 := \{\neg\, ?\varepsilon(k),\ ?\varepsilon(tl(k))\}$ and $\Delta_2 := \{\mathit{true}\}$, and

- $\vdash^{split}_{\Gamma,C_3} \langle \nabla_3, \Delta_3, (k,\epsilon) \succcurlyeq ((\ldots \bullet (hd(tl(k)) :: snd(split(tl(tl(k))))))), \mathbf{2})\rangle$
  for $C_3 := \{\neg\, ?\varepsilon(k),\ \neg\, ?\varepsilon(tl(k))\}$ and
  $\Delta_3 := \big\{\Delta^{1,\epsilon}_{tl,\epsilon}(tl(k)),\ \Delta^{1,\epsilon}_{split,\mathbf{2}}(tl(tl(k))),\ \Delta^{1,\mathbf{2}}_{snd,\epsilon}(split(tl(tl(k))))\big\}$.

$\bigvee \Delta_1$ is *false*, while $\bigvee \Delta_2$ is *true*. $\bigvee \Delta_3$ can again be simplified to *true*. The difference procedure $\Delta^{1,\epsilon}_{split,\mathbf{2}}$ in Figure 4.6 corresponds to the intuition that the sublist $e_2 :: e_4 :: \ldots$ of $k = e_1 :: e_2 :: e_3 :: e_4 :: \ldots$ is strictly smaller than k iff k is non-empty. ♢

4.3.3 Instantiating Argument-Bounded Functions

Example 4.20 (p. 96) presented several argument-bounded selectors. For instance, $hd : list[@A] \to @A$ is $(1, \mathbf{1}, \epsilon)$-argument-bounded. By Definition 4.16, argument-boundedness of hd extends to all instantiations $\theta(@A)$ of $@A$.

For example, for type substitution $\theta := \{@A/pair[\mathbb{N}, term[\tau_V, \tau_F]]\}$ and some list $k : list[pair[\mathbb{N}, term[\tau_V, \tau_F]]]$, $hd(k)$ contains at most as many *pair*-constructors as k. But furthermore, $hd(k)$ also contains at most as many $\mathbb{N}$-constructors as k; the same holds for the number of *term*-constructors. Thus the instance

$$\theta(hd) : list[pair[\mathbb{N}, term[\tau_V, \tau_F]]] \to pair[\mathbb{N}, term[\tau_V, \tau_F]]$$

of selector hd is also $(1, \mathbf{11}, \mathbf{1})$-argument-bounded and $(1, \mathbf{12}, \mathbf{2})$-argument-bounded. Similar considerations hold for the $(1, \mathbf{1}, \epsilon)$-argument-bounded procedure *last* (cf. Example 4.17 on p. 94). Apparently, any θ-instance of hd and *last* is $(1, \mathbf{1}\pi, \pi)$-argument-bounded for any $\pi \in Pos(\theta(@A))$.

The following lemma shows that θ-instances of selectors are argument-bounded:

Lemma 4.42. *Let $sel_j : \tau \to \tau_j$ be a $(1, h, \varrho)$-argument-bounded selector for $\tau = str[@A_1, \ldots, @A_k]$ and $h \in \{1, \ldots, k\}$. Let θ be a type substitution and $\pi \in Pos(\theta(@A_h))$. Then $\theta(sel_j)$ is $(1, h\pi, \varrho\pi)$-argument-bounded.*[8]

Proof. We may assume without loss of generality that θ is an arbitrary *grounding* type substitution (i. e., we consider any extension of a type substitution to a grounding type substitution). Let $q \in \mathcal{T}(\Sigma(P)^{\mathsf{c}})_{\theta(\tau)}$ be a value with $eval_P(c_{sel_j}[q]) = true$. Thus $q = cons(q_1, \ldots, q_n)$ for some $q_i \in \mathcal{T}(\Sigma(P)^{\mathsf{c}})_{\theta(\tau_i)}$, where $cons : \tau_1 \times \ldots \times \tau_n \to \tau$ is the constructor that sel_j belongs to.

[8] The corresponding difference procedure $\Delta^{1,h\pi}_{\theta(sel_j),\varrho\pi}$ for $\theta(sel_j)$ can be synthesized as described in Theorem 4.23 (p. 98) by appending type position π to h.

We conclude (†) $(j, \varrho) \in Occ_{@A_h}(cons)$ from $\tau_j\|_\varrho = \tau\|_h = @A_h$. Hence

$$\begin{aligned} \#(q, h\pi) &= \textstyle\sum_{(j',\pi'')\in Occ_{@A_h}(cons)} \#(q_{j'}, \pi''\pi) && \text{; by def. of } \# \\ &\geq \#(q_j, \varrho\pi) && \text{; by (†)} \\ &= \#(eval_P(sel_j(q)), \varrho\pi) && \text{; by def. of } eval_P \end{aligned}$$

as desired. □

Since the instantiation of a selector with a type substitution θ preserves argument-boundedness, θ-instances of argument-bounded procedures are argument-bounded as well: In Example 4.17 (p. 94) we proved that procedure *last* is argument-bounded using the fact that selectors *hd* and *tl* are argument-bounded. Since the θ-instances of *hd* and *tl* are also argument-bounded, this proof carries over to the θ-instance of procedure *last*.

Summing up, a θ-instance $\theta(f)$ of a (p, π, ϱ)-argument-bounded function f is $(p, \pi\pi', \varrho\pi')$-argument-bounded if $\tau_p\|_\pi$ is a type variable and $\pi' \in Pos(\theta(\tau_p\|_\pi))$.

4.3.4 Estimation Proofs in Termination Analysis

Now we are ready to state a termination criterion for procedures *without* second-order recursion. The formulas *(ii)* of Theorem 4.43 are *termination hypotheses* for procedure f; if these formulas are true, the procedure terminates. For simplicity, we consider only unary procedures f. We generalize the theorem in Section 4.5 to procedures with second-order recursion and arbitrary arity.

Theorem 4.43. *A procedure* `procedure` $f(x:\tau) : \tau' <= B_f^{\mathsf{rel}}$ *for a base type τ terminates if all procedures g with $f >^+_{uses} g$ terminate and if there is some $\pi \in Pos(\tau)$ such that each recursive call $f(t)$ in B_f^{rel} under some call context $C \in \mathcal{CL}(\Sigma, \mathcal{V})$ is a* direct *procedure call such that*

(i) $\vdash_{\Gamma,C} \langle \nabla, \Delta, (x, \pi) \succcurlyeq (t, \pi)\rangle$ *for some ∇, Δ, and*

(ii) $\forall x:\tau.\ \bigwedge C \to \left[\bigwedge \nabla \wedge \bigvee \Delta\right]$ *is true.*

Proof. We only sketch the proof, because Theorem 4.62 (p. 125) subsumes this theorem. Whenever $q \succ^\theta_f q'$, then $\#(q, \pi) > \#(q', \pi)$ by (i), (ii), and Theorem 4.35(2). Thus the size measure strictly decreases for recursive calls, so procedure f terminates by Lemma 4.1 (p. 85). □

Example 4.44. Procedure *qsort* (cf. Figure 4.7) terminates. For $\pi := \epsilon$, $C := \{\neg\, ?\varepsilon(k)\}$, and any g, $(k, \epsilon) \geqslant_{\Gamma,C} (filter(g, tl(k)), \epsilon)$ with $\bigwedge \nabla = true$ and $\bigvee \Delta = true$, cf. Example 4.30 (p. 102). ♢

```
procedure qsort(k : list[ℕ]) : list[ℕ] <=
if ?ε(k)
  then ε
  else qsort(filter(λn : ℕ. n ≤ hd(k), tl(k))) <> hd(k) ::
       qsort(filter(λn : ℕ. n > hd(k), tl(k)))
end
```

Figure 4.7: Implementation of Quicksort

Example 4.45. Procedure *termlist.size* terminates:

```
procedure termlist.size(k : list[term[@V, @F]]) : ℕ <=
if ?ε(k) then 0
  else if ?var(hd(k))
         then 1 + termlist.size(tl(k))
         else 1 + termlist.size(tl(k)) + termlist.size(args(hd(k)))
end   end
```

We choose $\pi := \mathbf{1}$ and consider the three recursive calls:

- $termlist.size(tl(k))$ under call context $C := \{\neg\, ?\varepsilon(k),\ ?var(hd(k))\}$:
  1. $\vdash_{\Gamma,C} \langle\{?::(k)\}, \{\Delta^{1,\mathbf{1}}_{tl,\mathbf{1}}(k)\}, (k,\mathbf{1}) \succcurlyeq (tl(k),\mathbf{1})\rangle$
  2. $\forall k : list[term[@V, @F]].$
     $\neg\, ?\varepsilon(k) \wedge ?var(hd(k)) \rightarrow \left[?::(k) \wedge \Delta^{1,\mathbf{1}}_{tl,\mathbf{1}}(k)\right]$
     is true, see Figure 4.3 (p. 98)
- $termlist.size(tl(k))$ under call context $C := \{\neg\, ?\varepsilon(k),\ \neg\, ?var(hd(k))\}$:
  1. as above
  2. $\forall k : list[term[@V, @F]].$
     $\neg\, ?\varepsilon(k) \wedge \neg\, ?var(hd(k)) \rightarrow \left[?::(k) \wedge \Delta^{1,\mathbf{1}}_{tl,\mathbf{1}}(k)\right]$
     is true, see Figure 4.3 (p. 98)
- $termlist.size(args(hd(k)))$ under call context $C := \{\neg\, ?\varepsilon(k),\ \neg\, ?var(hd(k))\}$:
  1. $\vdash_{\Gamma,C} \langle\nabla, \Delta, (k,\mathbf{1}) \succcurlyeq (args(hd(k)),\mathbf{1})\rangle$ for
     $\nabla := \{?::(k), ?apply(hd(k))\}$ and $\Delta := \{\Delta^{1,\mathbf{1}}_{hd,\epsilon}(k), \Delta^{1,\epsilon}_{args,\mathbf{1}}(hd(k))\}$
  2. $\forall k : list[term[@V, @F]].\ \neg\, ?\varepsilon(k) \wedge \neg\, ?var(hd(k)) \rightarrow$
     $\left[?::(k) \wedge ?apply(hd(k)) \wedge \left(\Delta^{1,\mathbf{1}}_{hd,\epsilon}(k) \vee \Delta^{1,\epsilon}_{args,\mathbf{1}}(hd(k))\right)\right]$
     is true, see Figure 4.3 (p. 98) ♢

```
procedure merge(k, l : list[ℕ]) : list[ℕ] <=
if ?ε(k)
  then l
  else if ?ε(l)
         then k
         else if hd(k) > hd(l)
                then hd(l) :: merge(k, tl(l))
                else hd(k) :: merge(tl(k), l)
end    end    end

procedure msort(k : list[ℕ]) : list[ℕ] <=
if ?ε(k)
  then ε
  else if ?ε(tl(k))
         then k
         else let divide := split(k) in
                merge(msort(fst(divide)), msort(snd(divide)))
end    end    end
```

Figure 4.8: Implementation of Mergesort

Example 4.46. Procedure *msort* (cf. Figure 4.8) terminates. (Here we assume termination of procedure *merge*. We prove termination of *merge* with a more powerful theorem in Example 4.67 on p. 128.)

We choose $\pi := \epsilon$ and consider the two recursive calls:

- $msort(fst(split(k)))$ under call context $C := \{\neg\, ?\varepsilon(k),\ \neg\, ?\varepsilon(tl(k))\}$:
  1. $\vdash_{\Gamma,C} \langle \nabla, \Delta, (k, \epsilon) \succcurlyeq (fst(split(k)), \epsilon)\rangle$ for $\nabla := \{?\bullet(split(k))\}$ and $\Delta := \{\Delta^{1,\mathbf{1}}_{fst,\epsilon}(split(k)), \Delta^{1,\epsilon}_{split,\mathbf{1}}(k)\}$
  2. $\forall k : list[\mathbb{N}].\ \neg\, ?\varepsilon(k) \wedge \neg\, ?\varepsilon(tl(k)) \rightarrow$ $\big[?\bullet(split(k)) \wedge \big(\Delta^{1,\mathbf{1}}_{fst,\epsilon}(split(k)) \vee \Delta^{1,\epsilon}_{split,\mathbf{1}}(k)\big)\big]$ is true, because $?\bullet(split(k))$ yields *true* (as $\bullet$ is the only data constructor of *pair*) and $\Delta^{1,\epsilon}_{split,\mathbf{1}}(k)$ returns *true* for lists with at least two elements (cf. Figure 4.6 on p. 112).
- $msort(snd(split(k)))$ under call context $C := \{\neg\, ?\varepsilon(k),\ \neg\, ?\varepsilon(tl(k))\}$:
  1. $\vdash_{\Gamma,C} \langle \nabla, \Delta, (k, \epsilon) \succcurlyeq (snd(split(k)), \epsilon)\rangle$ for $\nabla := \{?\bullet(split(k))\}$ and $\Delta := \{\Delta^{1,\mathbf{2}}_{snd,\epsilon}(split(k)), \Delta^{1,\epsilon}_{split,\mathbf{2}}(k)\}$
  2. $\forall k : list[\mathbb{N}].\ \neg\, ?\varepsilon(k) \wedge \neg\, ?\varepsilon(tl(k)) \rightarrow$ $\big[?\bullet(split(k)) \wedge \big(\Delta^{1,\mathbf{2}}_{snd,\epsilon}(split(k)) \vee \Delta^{1,\epsilon}_{split,\mathbf{2}}(k)\big)\big]$ is true, because $\Delta^{1,\epsilon}_{split,\mathbf{2}}(k)$ returns *true* for non-empty lists k (cf. Figure 4.6 on p. 112). ♢

Examples 4.45 and 4.46 cannot be solved by the original method in [86, 96], because a list is always measured by its *length* in [86, 96], which corresponds to the special case $\pi = \epsilon$ of our theorem (e. g., Example 4.44). Furthermore, neither *hd*, *args*, nor *split* are considered as argument-bounded in [86, 96] due to the non-parameterized size measure, so the difference procedures $\Delta^{1,\mathbf{1}}_{hd,\epsilon}$, $\Delta^{1,\epsilon}_{args,\mathbf{1}}$, $\Delta^{1,\epsilon}_{split,\mathbf{1}}$, and $\Delta^{1,\epsilon}_{split,\mathbf{2}}$ do not exist there.

So far we have extended the approach of [86, 96] by using a greater variety of size measures; a type position designates the component of a type that we wish to consider in the size estimations. The next section focuses on *second-order recursion*: We introduce the concept of *call-bounded* second-order procedures to facilitate termination proofs of procedures that are defined by second-order recursion.

4.4 Call-Bounded Second-Order Procedures

Call-bounded procedures g are well-behaved in the sense that they call their functional parameter only with arguments of a bounded size: Whenever $g(f, q) \rhd_f f(q')$, the size of q is a bound of the size of q'.

We consider only procedures g with two parameters in the following definition for readability reasons; the definitions and theorems generalize to procedures of arbitrary arity in a straightforward way, see Section 4.4.1 and Example 4.49 below.

Definition 4.47 (Call-bounded procedures)**.** *A procedure*

$$\texttt{procedure}\ g(f : \tau_1 \times \ldots \times \tau_m \to \tau',\ x : \tau) : \tau'' <= B^{\mathsf{rel}}_g$$

is (π, r, ϱ)-call-bounded *for* $\pi \in Pos(\tau)$, $r \in \{1, \ldots, m\}$, *and* $\varrho \in Pos(\tau_r)$ *iff* τ *is a base type with* $\tau\|_\pi = \tau_r\|_\varrho$ *such that for all grounding type substitutions* $\theta \in GndSubst_{\Omega(P)}(\tau_1, \ldots, \tau_m, \tau', \tau)$, *all values* $\mathsf{x} \in \mathbb{V}(P)_{\theta(\tau)}$, *all fresh functions* $\mathsf{f} \in \mathbb{V}(P)_{\theta(\tau_1 \times \ldots \times \tau_m \to \tau')}$, *and all values* q_i *with* $q_i \in \mathbb{V}(P)_{\theta(\tau_i)}$ *for* $i = 1, \ldots, m$,

$$g(\mathsf{f}, \mathsf{x}) \rhd_{\mathsf{f}} \mathsf{f}(q_1, \ldots, q_m) \implies \#_{\theta(\tau)}(\mathsf{x}, \pi) \geq \#_{\theta(\tau_r)}(q_r, \varrho)\,.$$

Example 4.48. Procedure *every* (cf. Figure 1.3 on p. 6) is $(\mathbf{1}, 1, \epsilon)$-call-bounded, because parameter p is only called with an argument $z : @A$ with $\#(k, \mathbf{1}) \geq \#(z, \epsilon)$. More formally, $\#(\mathsf{k}, \mathbf{1}) \geq \#(\mathsf{z}, \epsilon)$ whenever

$$every(\mathsf{p}, \mathsf{k}) \rhd_{\mathsf{p}} \mathsf{p}(\mathsf{z})\,.$$

For the same reason, procedure *filter* is $(\mathbf{1}, 1, \epsilon)$-call-bounded. ♢

Example 4.49. Procedure *foldl* (cf. Figure 1.4 on p. 7) is $(\mathbf{1}, 2, \epsilon)$-call-bounded wrt. parameter k, because $\#(\mathsf{k}, \mathbf{1}) \geq \#(\mathsf{b}, \epsilon)$ whenever

$$foldl(\mathsf{f}, \mathsf{x}, \mathsf{k}) \rhd_{\mathsf{f}} \mathsf{f}(\mathsf{a}, \mathsf{b})\,.$$ ♢

Lemma 4.50. *For each second-order procedure*

$$\texttt{procedure}\ \mathit{proc}(f : \tau_1 \times \ldots \times \tau_m \to \tau_f,\ x : \tau_x) : \tau_{\mathit{proc}}$$

which is (π, r, ϱ)-call-bounded, the associated quantification procedure

$$\begin{aligned}\texttt{procedure}\ \mathit{forall.proc}(&p : \tau_1 \times \ldots \times \tau_m \to \mathit{bool},\\ &f : \tau_1 \times \ldots \times \tau_m \to \tau_f,\\ &x : \tau_x) : \mathit{bool}\end{aligned}$$

is also (π, r, ϱ)-call-bounded wrt. p and f.

Proof. Lemma 3.9 (p. 75) asserts that *forall.proc* calls p and f only if *proc* already calls f. Thus call-boundedness of *proc* is passed on to *forall.proc*, both wrt. parameter p and wrt. parameter f. □

4.4.1 Proving Call-Boundedness of Procedures

We can easily identify many call-bounded procedures with the help of the following theorem:

Theorem 4.51. *A procedure*

$$\texttt{procedure}\ g(f : \tau_1 \times \ldots \times \tau_m \to \tau',\ x : \tau) : \tau'' <= B_g^{\mathsf{rel}}$$

is (π, r, ϱ)-call-bounded for $\pi \in Pos(\tau)$, $r \in \{1, \ldots, m\}$, and $\varrho \in Pos(\tau_r)$ if τ is a base type with $\tau\|_\pi = \tau_r\|_\varrho$ such that f occurs in B_g^{rel} only in direct function calls $f(t_1, \ldots, t_m)$ under some call context C such that

1. *$\vdash_{\Gamma,C} \langle \nabla, \Delta, (x, \pi) \succcurlyeq (t_r, \varrho)\rangle$ for some ∇, Δ and*
2. *$\forall f : \tau_1 \times \ldots \times \tau_m \to \tau',\ x : \tau.\ \bigwedge C \to \bigwedge \nabla$ is true*

or in direct recursive calls $g(f, t')$ under some call context C' such that

3. *$\vdash_{\Gamma,C'} \langle \nabla', \Delta', (x, \pi) \succcurlyeq (t', \pi)\rangle$ for some ∇', Δ' and*
4. *$\forall f : \tau_1 \times \ldots \times \tau_m \to \tau',\ x : \tau.\ \bigwedge C' \to \bigwedge \nabla'$ is true.*

Proof. Assume $g(\mathsf{f}, \mathsf{x}_0) \rhd_{\mathsf{f}} \mathsf{f}(q_1, \ldots, q_m)$ for a grounding type substitution $\theta \in \mathit{GndSubst}_{\Omega(P)}(\tau_1, \ldots, \tau_m, \tau)$, a value $\mathsf{x}_0 \in \mathbb{V}(P)_{\theta(\tau)}$, and a fresh function $\mathsf{f} \in \mathbb{V}(P)_{\theta(\tau_1 \times \ldots \times \tau_m \to \tau')}$. Then

$$g(\mathsf{f}, \mathsf{x}_0) \rhd g(\mathsf{f}, \mathsf{x}_1) \rhd \ldots \rhd g(\mathsf{f}, \mathsf{x}_n) \rhd \mathsf{f}(q_1, \ldots, q_m)$$

for values $\mathsf{x}_1, \ldots, \mathsf{x}_n \in \mathbb{V}(P)_{\theta(\tau)}$.

(i) We show $\#(\mathsf{x}_0, \pi) \geq \#(\mathsf{x}_n, \pi)$ by induction on n. This is trivial for the base case $n = 0$. For the step case, let $g(f, t')$ be the direct recursive call of g under call context C' in B_g^{rel} that corresponds to $g(\mathsf{f}, \mathsf{x}_n) \rhd g(\mathsf{f}, \mathsf{x}_{n+1})$; i. e., (†) $\mathsf{x}_{n+1} = \mathit{eval}_P(\sigma(t'))$ and (‡) $\mathit{eval}_P(\sigma(c')) = \mathit{true}$ for all $c' \in C'$, where $\sigma := \{f/\mathsf{f}, x/\mathsf{x}_n\}$. By (3), (4), (‡), and Theorem 4.35(1):

$$\begin{aligned}\#(\mathsf{x}_0, \pi) &\overset{\text{(IH)}}{\geq} \#(\mathsf{x}_n, \pi) = \#(\sigma(x), \pi) \\ &\geq \#(\mathit{eval}_P(\sigma(t')), \pi) \overset{(\dagger)}{=} \#(\mathsf{x}_{n+1}, \pi)\,.\end{aligned}$$

(ii) We show $\#(\mathsf{x}_n, \pi) \geq \#(q_r, \varrho)$: Let $f(t_1, \ldots, t_m)$ be the function call under call context C in B_g^{rel} that corresponds to $g(\mathsf{f}, \mathsf{x}_n) \rhd \mathsf{f}(q_1, \ldots, q_m)$; i. e., (†) $q_i = \mathit{eval}_P(\sigma(t_i))$ for all $i = 1, \ldots, m$ and (‡) $\mathit{eval}_P(\sigma(c)) = \mathit{true}$ for all $c \in C$, where $\sigma := \{f/\mathsf{f}, x/\mathsf{x}_n\}$. By (1), (2), (‡), and Theorem 4.35(1):

$$\#(\mathsf{x}_n, \pi) = \#(\sigma(x), \pi) \geq \#(\mathit{eval}_P(\sigma(t_r)), \varrho) \overset{(\dagger)}{=} \#(q_r, \varrho)\,.$$

(iii) From (i) and (ii), we conclude $\#(\mathsf{x}_0, \pi) \geq \#(q_r, \varrho)$ as desired. □

Note that Theorem 4.51 does *not* require procedure g to be terminating: Definition 4.47 only requires that for all calls $g(\mathsf{f}, \mathsf{x}) \rhd_{\mathsf{f}} \mathsf{f}(q_1, \ldots, q_m)$, the argument q_r of f is smaller than x. For this definition, it does not matter if there are infinitely many such calls. Of course, in practice we are interested in *terminating* call-bounded procedures such as *every*.

Example 4.52. Procedure *map* (cf. Figure 1.3 on p. 6) is $(\mathbf{1}, 1, \epsilon)$-call-bounded:

1. $\vdash_{\Gamma, C} \langle\{?::(k)\}, \{\Delta^{1,\mathbf{1}}_{hd,\epsilon}(k)\}, (k, \mathbf{1}) \succcurlyeq (\mathit{hd}(k), \epsilon)\rangle$ for $C = \{\neg\, ?\varepsilon(k)\}$
2. $\forall f : @A \to @B, k : \mathit{list}[@A].\ \neg\, ?\varepsilon(k) \to ?::(k)$ is true
3. $\vdash_{\Gamma, C'} \langle\{?::(k)\}, \{\Delta^{1,\mathbf{1}}_{tl,\mathbf{1}}(k)\}, (k, \mathbf{1}) \succcurlyeq (\mathit{tl}(k), \mathbf{1})\rangle$ for $C' = \{\neg\, ?\varepsilon(k)\}$
4. $\forall f : @A \to @B, k : \mathit{list}[@A].\ \neg\, ?\varepsilon(k) \to ?::(k)$ is true ♢

Example 4.53. Procedure *every* (cf. Figure 1.3 on p. 6) is $(\mathbf{1}, 1, \epsilon)$-call-bounded:

1. $\vdash_{\Gamma, C} \langle\{?::(k)\}, \{\Delta^{1,\mathbf{1}}_{hd,\epsilon}(k)\}, (k, \mathbf{1}) \succcurlyeq (\mathit{hd}(k), \epsilon)\rangle$ for $C = \{\neg\, ?\varepsilon(k)\}$
2. $\forall p : @A \to \mathit{bool}, k : \mathit{list}[@A].\ \neg\, ?\varepsilon(k) \to ?::(k)$ is true

3. $\vdash_{\Gamma,C'} \langle\{?::(k)\}, \{\Delta^{1,\mathbf{1}}_{tl,\mathbf{1}}(k)\}, (k,\mathbf{1}) \succcurlyeq (tl(k),\mathbf{1})\rangle$ for $C' = \{\neg\, ?\varepsilon(k),\ p(hd(k))\}$

4. $\forall p : @A \to bool, k : list[@A].\ \neg\, ?\varepsilon(k) \wedge p(hd(k)) \to ?::(k)$ is true

Similarly, procedures *some* and *filter* are $(\mathbf{1}, 1, \epsilon)$-call-bounded. ♢

The following example demonstrates how Theorem 4.51 easily generalizes to procedures with more than two parameters:

Example 4.54. Procedure *foldl* (cf. Figure 1.4 on p. 7) is $(\mathbf{1}, 2, \epsilon)$-call-bounded wrt. parameter k:

1. $\vdash_{\Gamma,C} \langle\{?::(k)\}, \{\Delta^{1,\mathbf{1}}_{hd,\epsilon}(k)\}, (k,\mathbf{1}) \succcurlyeq (hd(k),\epsilon)\rangle$ for $C = \{\neg\, ?\varepsilon(k)\}$

2. $\forall f : @A \times @B \to @A,\ x : @A,\ k : list[@B].\ \neg\, ?\varepsilon(k) \to ?::(k)$ is true

3. $\vdash_{\Gamma,C'} \langle\{?::(k)\}, \{\Delta^{1,\mathbf{1}}_{tl,\mathbf{1}}(k)\}, (k,\mathbf{1}) \succcurlyeq (tl(k),\mathbf{1})\rangle$ for $C' = \{\neg\, ?\varepsilon(k)\}$

4. $\forall f : @A \times @B \to @A,\ x : @A,\ k : list[@B].\ \neg\, ?\varepsilon(k) \to ?::(k)$ is true

Similarly, procedures *foldr* and *rev_itlist* are $(\mathbf{1}, 1, \epsilon)$-call-bounded wrt. parameter k. ♢

The next example illustrates that the estimation proofs required for Theorem 4.51 quickly fail if a procedure is not (π, r, ϱ)-call-bounded.[9] This means that in practice it is feasible to just check call-boundedness of a procedure for all combinations of π, r, and ϱ.

Example 4.55. Procedure *foldl* is *not* $(\epsilon, 1, \epsilon)$-call-bounded wrt. parameter x:

3. $\vdash_{\Gamma,C'} \langle\nabla', \Delta', (x,\epsilon) \succcurlyeq (f(x, hd(k)),\epsilon)\rangle$ for $C' = \{\neg\, ?\varepsilon(k)\}$

cannot be shown, because no estimation rule is applicable to $\langle\emptyset, \emptyset, \{(x,\epsilon) \succcurlyeq (f(x, hd(k)),\epsilon)\}\rangle$. This makes sense, because f can be instantiated with "+", and $x \not\succcurlyeq x + hd(k)$ in general. ♢

4.4.2 Instantiating Call-Bounded Procedures

Since Theorem 4.51 uses the estimation calculus to prove that a procedure $g : (\tau_1 \times \ldots \times \tau_m \to \tau') \times \tau \to \tau''$ is (π, r, ϱ)-call-bounded, each instance $\theta(g)$ is $(\pi\pi', r, \varrho\pi')$-call-bounded if $\tau\|_\pi = @A = \tau_r\|_\varrho$ for some type variable $@A$, a type substitution θ, and an arbitrary type position $\pi' \in Pos(\theta(@A))$, because the argument-bounded functions that procedure f calls can be instantiated accordingly, see Section 4.3.3.

[9] In general, deductions in the estimation calculus are quite short. As soon as no estimation rule is applicable to some element of E, a proof attempt fails (unless backtracking is possible).

Example 4.56. For $\theta := \{@A/\mathit{list}[@C]\}$, the θ-instance

$$\theta(\mathit{map}) : (\mathit{list}[@C] \to @B) \times \mathit{list}[\mathit{list}[@C]] \to \mathit{list}[@B]$$

of procedure *map* is $(\mathbf{11}, 1, \mathbf{1})$-call-bounded. Intuitively, any $l : \mathit{list}[@C]$ that $\theta(\mathit{map})$ applies its first argument to contains at most as many $@C$'s as $k : \mathit{list}[\mathit{list}[@C]]$. In Example 4.59 (p. 124) we use such a call-bounded instance of *map* to show call-boundedness of another procedure. ♢

4.4.3 Generalized Detection of Call-Bounded Procedures

Theorem 4.51 handles the frequently occurring special case of Definition 4.47 where $g(f, x) \rhd g(f, x_1) \rhd \ldots \rhd g(f, x_n) \rhd f(z)$; i. e., the first-order parameter f is either called directly or passed to the recursive call $g(f, t')$ without modification. There are three obvious generalizations of Theorem 4.51:

Allow g to modify f in recursive calls. Procedure g is also (π, r, ϱ)-call-bounded if the recursive calls under call context C' in B_g^{rel} have the form $g(\lambda y_1 : \tau_1, \ldots, y_m : \tau_m.\, t'', t')$ such that for each call $f(t_1, \ldots, t_m)$ in t'' under call context C'':

5. $\vdash_{\Gamma, C' \cup C''} \langle \nabla'', \Delta'', (y_r, \varrho) \succcurlyeq (t_r, \varrho) \rangle$ for some ∇'', Δ'' and
6. $\forall f : \tau_1 \times \ldots \times \tau_m \to \tau', x : \tau, y_1 : \tau_1, \ldots, y_m : \tau_m.\ \bigwedge C' \wedge \bigwedge C'' \to \bigwedge \nabla''$ is true

in addition to requirements (3) and (4). The idea is that the λ-function can reduce the size of the r-th parameter t_r of f further: $(x, \pi) \geqslant_{\#} (y_r, \varrho) \geqslant_{\#} (t_r, \varrho)$.

Example 4.57. Procedure *foo* in Figure 4.9 is $(\mathbf{1}, 1, \epsilon)$-call-bounded: Requirements (3) and (4) are satisfied, so $(k, \mathbf{1}) \geqslant_{\#} (\mathit{tl}(k), \mathbf{1})$, and in addition:

5. $\vdash_{\Gamma, C' \cup C''} \langle \nabla'', \Delta'', (y, \epsilon) \succcurlyeq (^{-}(y), \epsilon) \rangle$ for $C' := \{\neg\, ?\varepsilon(k), p(\mathit{hd}(k))\}$, $C'' := \{\neg\, ?\mathtt{0}(y)\}$, $\nabla'' := \{?^{+}(y)\}$, $\Delta'' := \{\Delta^{1,\epsilon}_{-_{(\ldots),\epsilon}}(y)\}$ and
6. $\forall p : \mathbb{N} \to \mathit{bool}, k : \mathit{list}[\mathbb{N}], y : \mathbb{N}.\ \neg\, ?\varepsilon(k) \wedge p(\mathit{hd}(k)) \wedge \neg\, ?\mathtt{0}(y) \to ?^{+}(y)$ is obviously true. ♢

Allow g to pass f to another call-bounded procedure $g' \neq g$. Procedure g is also (π, r, ϱ)-call-bounded if B_g^{rel} contains a subterm $g'(f, t')$ under call context C' such that

7. g' is (π', r', ϱ')-call-bounded for some π', r', and $\varrho' = \varrho$,
8. $\vdash_{\Gamma, C'} \langle \nabla', \Delta', (x, \pi) \succcurlyeq (t', \pi') \rangle$ for some ∇', Δ' and
9. $\forall f : \tau_1 \times \ldots \times \tau_m \to \tau',\ x : \tau.\ \bigwedge C' \to \bigwedge \nabla'$ is true.

This generalization is sound, because $\varrho' = \varrho$ ensures that g' calls f only with arguments y_r with $(x, \pi) \geqslant_{\#} (t', \pi') \geqslant_{\#} (y_r, \varrho)$.

```
procedure foo(p:ℕ → bool, k:list[ℕ]) : bool <=
if ?ε(k)
  then true
  else if p(hd(k))
         then foo(λy:ℕ. if ?0(y) then false else p(⁻(y)) end, tl(k))
         else false
end    end

procedure index.of(x:@A, k:list[@A]) : ℕ <=
if ?ε(k)
  then 0
  else if hd(k) = x
         then 1
         else ⁺(index.of(x, tl(k)))
end    end

procedure find.root(f:@A → ℕ, k:list[@A]) : ℕ <=
index.of(0, map(f, k))
```

Figure 4.9: Slightly more complicated call-bounded procedures

Example 4.58. Procedure *find.root* in Figure 4.9 is $(\mathbf{1}, 1, \epsilon)$-call-bounded, because it just passes f to the $(\mathbf{1}, 1, \epsilon)$-call-bounded procedure *map*. (Procedure *find.root* computes the index of the first element of k that is a root of function f. It returns 0 if $f(x) \neq 0$ for all $x \in k$.) ♢

Allow g to pass f to an indirect recursive call. Procedure g is also (π, r, ϱ)-call-bounded if its body contains an *indirect* recursive call via a call-bounded procedure g':

Let $g'(\lambda y_1 : \tau'_1, \ldots, y_n : \tau'_n.\, t'',\, t')$ be a procedure call under some call context C' in B_g^{rel} such that t'' contains a call $g(f, t)$ under some call context C''. Then procedure g is (π, r, ϱ)-call-bounded if

10. procedure g' is (π', r', ϱ')-call-bounded for some π', r', and ϱ',
11. $\vdash_{\Gamma, C'} \langle \nabla', \Delta', (x, \pi) \succcurlyeq (t', \pi') \rangle$ for some ∇', Δ',
12. $\vdash_{\Gamma, C' \cup C''} \langle \nabla'', \Delta'', (y_{r'}, \varrho') \succcurlyeq (t, \pi) \rangle$ for some ∇'', Δ'',
13. $\forall f : \tau_1 \times \ldots \times \tau_m \to \tau',\ x : \tau.\ \bigwedge C' \to \bigwedge \nabla'$ is true, and
14. $\forall f : \tau_1 \times \ldots \times \tau_m \to \tau',\ x : \tau,\ y_1 : \tau'_1, \ldots, y_n : \tau'_n.\ \bigwedge C' \wedge \bigwedge C'' \to \bigwedge \nabla''$ is true.

This generalization is sound, because $(x, \pi) \geqslant_{\#} (t', \pi') \geqslant_{\#} (y_{r'}, \varrho') \geqslant_{\#} (t, \pi)$ ensures that the second argument of g (the "bound" x) does not increase in indirect recursive calls.

```
structure tree[@A] <=
  leaf(val : @A),
  branch(children : list[tree[@A]])

procedure treemap(f : @A → @A, t : tree[@A]) : tree[@A] <=
case t of
  leaf     : leaf(f(val(t)))
  branch : branch(map(λs : tree[@A]. treemap(f, s), children(t)))
end
```

Figure 4.10: Data structure definition $tree[@A]$ and procedure *treemap*

Example 4.59. Procedure *treemap* (cf. Figure 4.10) passes f to the indirect recursive call $treemap(f, s)$. Since the instance

$$\theta(map) : (tree[@A] \to tree[@A]) \times list[tree[@A]] \to list[tree[@A]]$$

for $\theta := \{@A/tree[@A]\}$ of *map* is $(\mathbf{11}, 1, \mathbf{1})$-call-bounded (cf. Example 4.56 on p. 122), s is bounded by $children(t)$, so

$$(t, \mathbf{1}) \geqslant_{\#} (children(t), \mathbf{11}) \geqslant_{\#} (s, \mathbf{1})\,. \qquad \diamondsuit$$

Since quantification procedures $forall.str_h$ are constructed solely from argument-bounded selectors (and quantification procedures $forall.str'_i$) according to Definition 3.1 (p. 65), they are call-bounded by construction:

Lemma 4.60. *For each type constructor str of arity k, quantification procedure $forall.str_h$ is $(h, 1, \epsilon)$-call-bounded for $h = 1, \ldots, k$.*

Proof. The proof is by induction wrt. $>_{uses}$: If $forall.str_h \not>_{uses} forall.str'_i$ for all $str' \neq str$, then we can directly apply Theorem 4.51 (p. 119) for the direct function calls $p(sel_j(x))$ and the direct recursive calls $forall.str_h(p, sel_j(x))$. Otherwise, we use the generalization from above, because the indirect calls of p occur via a call-bounded procedure $forall.str'_i$. □

Example 4.61. The quantification procedures in Figures 3.1 (p. 66) and 3.2 (p. 67) are call-bounded. This is obvious for *forall.list* (because it is equivalent to procedure *every*) and for $forall.pair_1$ and $forall.pair_2$ (because they are not defined recursively).

Procedures $forall.term_1$ and $forall.term_2$ require the generalizations discussed above. For instance, $forall.term_1$ uses the $(\mathbf{11}, 1, \mathbf{1})$-call-bounded instance

$$\theta(forall.list) : (term[@\,V, @F] \to bool) \times list[term[@\,V, @F]] \to bool$$

of *forall.list* for $\theta := \{@A/term[@V, @F]\}$. Therefore

$$(t, \mathbf{1}) \geqslant_{\#} (args(t), \mathbf{11}) \geqslant_{\#} (s, \mathbf{1})$$

as desired. Procedure $forall.term_2$ is call-bounded, because $\theta(forall.list)$ is also $(\mathbf{12}, 1, \mathbf{2})$-call-bounded, so

$$(t, \mathbf{2}) \geqslant_{\#} (args(t), \mathbf{12}) \geqslant_{\#} (s, \mathbf{2})$$

as desired. ♢

4.5 Proving Termination of Procedures

The concept of call-bounded procedures facilitates automated termination proofs of procedures that pass themselves to a call-bounded second-order procedure: In the following theorem, the arguments t of *direct* recursive calls need to decrease, cf. requirements *(1)* and *(2)*. *Indirect* recursive calls need to occur via a call-bounded procedure g, cf. *(3)*. This procedure g must be called with a bounding argument t' that is strictly smaller than the argument x of f, cf. *(4)* and *(5)*.

Theorem 4.62. *A procedure* $f(x:\tau) : \tau' <= B_f^{\mathsf{rel}}$ *for a base type* τ *terminates if all procedures* g *with* $f >^{+}_{uses} g$ *terminate and if there is some* $\pi \in Pos(\tau)$ *such that for each* direct *recursive call* $f(t)$ *in* B_f^{rel} *under some call context* $C \in \mathcal{CL}(\Sigma, \mathcal{V})$

1. $\vdash_{\Gamma,C} \langle \nabla, \Delta, (x, \pi) \succcurlyeq (t, \pi) \rangle$ *for some* ∇, Δ*, and*
2. $\forall x:\tau.\ \bigwedge C \to \left[\bigwedge \nabla \wedge \bigvee \Delta\right]$ *is true*

and for each indirect *recursive call* $g(f, t')$ *in* B_f^{rel} *under some call context* C'

3. *procedure* g *is* $(\pi', 1, \pi)$*-call-bounded for some* π'*,*
4. $\vdash_{\Gamma,C'} \langle \nabla', \Delta', (x, \pi) \succcurlyeq (t', \pi') \rangle$ *for some* ∇', Δ'*, and*
5. $\forall x:\tau.\ \bigwedge C' \to \left[\bigwedge \nabla' \wedge \bigvee \Delta'\right]$ *is true.*

Proof (informal sketch). The idea of the theorem is to ensure that the argument of a recursive call is smaller than x. For direct recursive calls $f(t)$, we have $(x, \pi) \geqslant_{\#} (t, \pi)$ by (1) and this inequality is strict due to (2). For indirect recursive calls $g(f, t')$, we have $(x, \pi) \geqslant_{\#} (t', \pi') \geqslant_{\#} (x', \pi)$ for the argument x' that f is called with by (4) and (3). Inequality $(x, \pi) \geqslant_{\#} (t', \pi')$ is strict due to (5). □

Proof (formal). We show that the requirements of Lemma 4.1 (p. 85) are satisfied. Let $\theta \in GndSubst_{\Omega(P)}(\tau)$ and $q, q' \in \mathbb{V}(P)_{\theta(\tau)}$ with $q \succ^{\theta}_{f} q'$. We show that $\#(q, \pi) > \#(q', \pi)$; i. e., we use the measure function m_θ defined by $m_\theta(q) := \#_{\theta(\tau)}(q)$.

Case $f(q) \rhd f(q')$: Let $f(t)$ be the corresponding direct recursive call; i. e., $q' = eval_P(t[x/q])$ and $eval_P(c[x/q]) = true$ for all $c \in C$. By (1), (2), and Theorem 4.35(2) we have indeed:

$$\#(q, \pi) = \#(eval_P(x[x/q]), \pi) > \#(eval_P(t[x/q]), \pi) = \#(q', \pi)$$

Case $f(q) \rhd g(f, q'') \rhd_f f(q')$: Let $g(f, t')$ be the call that corresponds to $f(q) \rhd g(f, q'')$; i. e., $q'' = eval_P(t'[x/q])$ and $eval_P(c'[x/q]) = true$ for all $c' \in C'$. By (4), (5), and Theorem 4.35(2) we have:

$$\#(q, \pi) = \#(eval_P(x[x/q]), \pi) > \#(eval_P(t'[x/q]), \pi') = \#(q'', \pi')$$

By (3) and $g(f, q'') \rhd_f f(q')$ we get:

$$\#(q'', \pi') \geq \#(q', \pi)$$

Hence $\#(q, \pi) > \#(q', \pi)$. □

Example 4.63. Procedure *groundterm* (cf. Figure 1.5 on p. 9) terminates: For $\pi := \epsilon$ and $C' := \{?apply(t)\}$:

3. procedure *every* is $(\mathbf{1}, 1, \epsilon)$-call-bounded, see Example 4.53 (p. 120)
4. $\vdash_{\Gamma, C'} \langle \{?apply(t)\}, \{\Delta^{1,\epsilon}_{args,\mathbf{1}}(t)\}, (t, \epsilon) \succcurlyeq (args(t), \mathbf{1}) \rangle$
5. $\forall t : term[@V, @F].\ ?apply(t) \to ?apply(t) \wedge \Delta^{1,\epsilon}_{args,\mathbf{1}}(t)$ is true, see Figure 4.3 (p. 98).

♢

Generalization: Allow f to occur in a λ-expression. Requirements (3)–(5) of Theorem 4.62 can be generalized so that the indirect recursive call may be nested in a λ-expression: If B^{rel}_f contains a call

$$g(\lambda y_1 : \tau_1, \ldots, y_m : \tau_m.\, t'',\ t')$$

under some call context C' and if t'' contains a call $f(t)$ under some call context C'', then the following requirements ensure termination of procedure f:

3′. procedure g is (π', r, ϱ)-call-bounded for some π', r, and ϱ,

4′. $\vdash_{\Gamma, C'} \langle \nabla', \Delta', (x, \pi) \succcurlyeq (t', \pi') \rangle$ for some ∇', Δ',

5′. $\vdash_{\Gamma, C' \cup C''} \langle \nabla'', \Delta'', (y_r, \varrho) \succcurlyeq (t, \pi) \rangle$ for some ∇'', Δ'',

6′. $\forall x : \tau.\ \bigwedge C' \to \bigwedge \nabla'$ is true,

7′. $\forall x : \tau, y_1 : \tau_1, \ldots, y_m : \tau_m.\ \bigwedge C' \wedge \bigwedge C'' \to \bigwedge \nabla''$ is true, and

8′. $\forall x : \tau, y_1 : \tau_1, \ldots, y_m : \tau_m.\ \bigwedge C' \wedge \bigwedge C'' \to \bigvee (\Delta' \cup \Delta'')$ is true.

```
procedure termsize(t : term[@V, @F]) : ℕ <=
case t of
  var   : 1,
  apply : foldl(λn : ℕ, s : term[@V, @F]. n + termsize(s), 1, args(t))
end
```

Figure 4.11: Alternative implementation of procedure *termsize*

This generalization is sound, because

$$(x, \pi) \geqslant_{\#} (t', \pi') \geqslant_{\#} (y_r, \varrho) \geqslant_{\#} (t, \pi)$$

due to (4′), (3′), and (5′), where (6′) and (7′) ensure the validity of the application of Theorem 4.35 (p. 108). The first or the third inequality is strict due to (8′).

Example 4.64. Procedure *termsize* (cf. Figure 4.11) counts the variable symbols and function symbols occurring in a term. This procedure terminates, because for $\pi := \epsilon$, $C' := \{?apply(t)\}$, and $C'' := \emptyset$ we get:

3′. procedure *foldl* is $(\mathbf{1}, 2, \epsilon)$-call-bounded, see Example 4.54 (p. 121),

4′. $\vdash_{\Gamma, C'} \langle \{?apply(t)\}, \{\Delta^{1,\epsilon}_{args,\mathbf{1}}(t)\}, (t, \epsilon) \succcurlyeq (args(t), \mathbf{1}) \rangle$,

5′. $\vdash_{\Gamma, C' \cup C''} \langle \emptyset, \emptyset, (s, \epsilon) \succcurlyeq (s, \epsilon) \rangle$,

6′. $\forall t : term[@V, @F].\ ?apply(t) \to ?apply(t)$ is true,

7′. $\forall t, s : term[@V, @F], n : \mathbb{N}.\ ?apply(t) \to true$ is true, and

8′. $\forall t, s : term[@V, @F], n : \mathbb{N}.\ ?apply(t) \to \Delta^{1,\epsilon}_{args,\mathbf{1}}(t)$ is true, see Figure 4.3 (p. 98). ♢

Generalization to arbitrary arity. Theorem 4.62 can be generalized in a straightforward way from a single parameter $x : \tau$ to n parameters $x_1 : \tau_1, \ldots, x_n : \tau_n$. After selecting an index $p \in \{1, \ldots, n\}$ of a parameter and a type position $\pi \in Pos(\tau_p)$, one uses x_p instead of x in the estimation proofs. Instead of argument t of a recursive call $f(t)$, one uses argument t_p of a recursive call $f(t_1, \ldots, t_n)$. Thus the additional $n - 1$ parameters and arguments are simply ignored.

Example 4.65. Procedure *map* (cf. Figure 1.3 on p. 6) terminates. For $p := 2$, $\pi := \epsilon$, and $C := \{\neg\, ?\varepsilon(k)\}$:

1. $\vdash_{\Gamma, C} \langle \emptyset, \{true\}, (k, \epsilon) \succcurlyeq (tl(k), \epsilon) \rangle$ and
2. $\forall f : @A \to @B,\ k : list[@A].\ \neg\, ?\varepsilon(k) \to true$ is true. ♢

Example 4.66. Procedure *subterm* (cf. Figure 1.5 on p. 9) terminates: For $p := 2$, $\pi := \epsilon$, $C' := \{r \neq t,\ ?apply(t)\}$, and $C'' := \emptyset$:

3′. procedure *some* is $(\mathbf{1}, 1, \epsilon)$-call-bounded, see Example 4.53 (p. 120),

4′. $\vdash_{\Gamma, C'} \langle \{?apply(t)\}, \{\Delta^{1,\epsilon}_{args,\mathbf{1}}(t)\}, (t, \epsilon) \succcurlyeq (args(t), \mathbf{1})\rangle$,

5′. $\vdash_{\Gamma, C' \cup C''} \langle \emptyset, \emptyset, (s, \epsilon) \succcurlyeq (s, \epsilon)\rangle$,

6′. $\forall r, t : term[@V, @F].\ r \neq t \wedge ?apply(t) \to ?apply(t)$ is true,

7′. $\forall r, t, s : term[@V, @F].\ r \neq t \wedge ?apply(t) \to true$ is true, and

8′. $\forall r, t, s : term[@V, @F].\ r \neq t \wedge ?apply(t) \to \Delta^{1,\epsilon}_{args,\mathbf{1}}(t)$ is true, see Figure 4.3 (p. 98). ♢

Similarly to [86, 96], it is of course also possible to consider several parameters at once. Instead of a single index p and a single type position π one then considers a list $(p_1, \pi_1), \ldots, (p_k, \pi_k)$ of indices $p_i \in \{1, \ldots, n\}$ and type positions $\pi_i \in Pos(\tau_{p_i})$. The size measures can either be combined lexicographically or by summing them up:

- For a lexicographic combination, each recursive call $f(t_1, \ldots, t_n)$ needs to satisfy $\#(x_{p_i}, \pi_i) \geq \#(t_{p_i}, \pi_i)$ for all $i = 1, \ldots, j$ and some $j \in \{1, \ldots, k\}$ as well as $\#(x_{p_j}, \pi_j) > \#(t_{p_j}, \pi_j)$.
- For a sum combination, each recursive call $f(t_1, \ldots, t_n)$ needs to satisfy $\#(x_{p_i}, \pi_i) \geq \#(t_{p_i}, \pi_i)$ for all $i = 1, \ldots, k$ and $\#(x_{p_j}, \pi_j) > \#(t_{p_j}, \pi_j)$ for at least one $j \in \{1, \ldots, k\}$, which ensures that $\sum_{i=1}^{k} \#(x_{p_i}, \pi_i) > \sum_{i=1}^{k} \#(t_{p_i}, \pi_i)$ gets smaller for the recursive call.

Example 4.67. Procedure *merge* (cf. Figure 4.8 on p. 117) terminates: We use a sum combination of the size measures of parameters $(p_1, \pi_1) := (1, \epsilon)$ and $(p_2, \pi_2) := (2, \epsilon)$. There are two recursive calls:

- $merge(k, tl(l))$ under call context $C := \{\neg ?\varepsilon(k),\ \neg ?\varepsilon(l),\ hd(k) > hd(l)\}$:
  1. $\vdash_{\Gamma, C} \langle \nabla_1, \Delta_1, (k, \epsilon) \succcurlyeq (k, \epsilon)\rangle$ for $\nabla_1 := \emptyset$ and $\Delta_1 := \emptyset$;
     $\vdash_{\Gamma, C} \langle \nabla_2, \Delta_2, (l, \epsilon) \succcurlyeq (tl(l), \epsilon)\rangle$ for $\nabla_2 := \emptyset$ and $\Delta_2 := \{true\}$
  2. $\forall k, l : list[\mathbb{N}].\ \neg ?\varepsilon(k) \wedge \neg ?\varepsilon(l) \wedge hd(k) > hd(l) \to true$ is true
- $merge(tl(k), l)$ under call context $C := \{\neg ?\varepsilon(k),\ \neg ?\varepsilon(l),\ \neg\, hd(k) > hd(l)\}$:
  1. $\vdash_{\Gamma, C} \langle \nabla_1, \Delta_1, (k, \epsilon) \succcurlyeq (tl(k), \epsilon)\rangle$ for $\nabla_1 := \emptyset$ and $\Delta_1 := \{true\}$;
     $\vdash_{\Gamma, C} \langle \nabla_2, \Delta_2, (l, \epsilon) \succcurlyeq (l, \epsilon)\rangle$ for $\nabla_2 := \emptyset$ and $\Delta_2 := \emptyset$
  2. $\forall k, l : list[\mathbb{N}].\ \neg ?\varepsilon(k) \wedge \neg ?\varepsilon(l) \wedge \neg\, hd(k) > hd(l) \to true$ is true ♢

Lemma 4.68. *For each type constructor str of arity k, quantification procedure forall.str$_h$ terminates.*

Proof. The proof is by induction wrt. $>_{uses}$: If *forall.str*$_h \not>_{uses}$ *forall.str*$'_i$ for all *str*$' \neq$ *str*, then we can apply Theorem 4.43 (p. 115), because there are no indirect recursive calls. The estimation proof of $(x, \epsilon) >_{\#} (sel_j(x), \epsilon)$ is trivial by using the fact that $sel_j : str[@A_1, \ldots, @A_k] \to str[@A_1, \ldots, @A_k]$ is $(1, \epsilon, \epsilon)$-argument-bounded. Otherwise, we use the generalization of Theorem 4.62 (p. 125), because each indirect recursive call of *forall.str*$_h$ occurs via a call-bounded procedure *forall.str*$'_i$ (cf. Lemma 4.60 on p. 124). □

Example 4.69. Procedure *forall.list* (cf. Figure 3.1 on p. 66) terminates: For $\pi := \epsilon$ and the recursive call under call context $C := \{?::(k), p(hd(k))\}$ we get

1. $\vdash_{\Gamma,C} \langle \emptyset, \{true\}, (k, \epsilon) \succcurlyeq (tl(k), \epsilon) \rangle$ and
2. $\forall p : @A \to bool,\ k : list[@A].\ ?::(k) \wedge p(hd(k)) \to true$ is true. ♢

Example 4.70. Procedures *forall.term*$_1$ and *forall.term*$_2$ (cf. Figure 3.2 on p. 67) terminate. We show the termination proof for *forall.term*$_1$; the termination proof for *forall.term*$_2$ is analogous. We measure the size wrt. parameter t and type position $\pi := \epsilon$. For the indirect recursive call under call contexts $C' := \{?apply(t)\}$ and $C'' := \emptyset$ we get:

3′. procedure *forall.list* is $(\mathbf{1}, 1, \epsilon)$-call-bounded, cf. Example 4.61 (p. 124),

4′. $\vdash_{\Gamma,C'} \langle \{?apply(t)\}, \{\Delta^{1,\epsilon}_{args,\mathbf{1}}(t)\}, (t, \epsilon) \succcurlyeq (args(t), \mathbf{1}) \rangle$,

5′. $\vdash_{\Gamma,C' \cup C''} \langle \emptyset, \emptyset, (s, \epsilon) \succcurlyeq (s, \epsilon) \rangle$,

6′. $\forall t : term[@V, @F].\ ?apply(t) \to ?apply(t)$ is true,

7′. $\forall t, s : term[@V, @F].\ ?apply(t) \to true$ is true, and

8′. $\forall t, s : term[@V, @F].\ ?apply(t) \to \Delta^{1,\epsilon}_{args,\mathbf{1}}(t)$ is true, see Figure 4.3 (p. 98). ♢

4.6 Summary

To prove termination of a procedure $f : \tau_1 \times \ldots \times \tau_n \to \tau$, we show that some measure $m : \tau_1 \times \ldots \times \tau_n \to \mathbb{N}$ decreases for the recursive calls.

In interactive termination analysis, this measure is provided by the user. Our approach then employs the quantification procedures from Chapter 3 to generate termination hypotheses for procedure f. If these termination hypotheses are true, the measure decreases for both direct and indirect recursive calls.

In automated termination analysis, the measure is given by a uniformly defined size measure. The size measure # computes the structural size of a value $q \in \mathbb{V}(P)_\tau$ of a ground base type τ by counting certain data constructors in q. It is parameterized by a type position π to specify the type component $\tau\|_\pi$ of τ whose data constructors in q are to be counted.

To automate the proof that the size measure decreases, we use the *estimation calculus*. A so-called *estimation proof* establishes a $\geq$-estimation of the size measure. Furthermore, an estimation proof yields a difference equivalent $\bigvee \Delta$ that is true iff the $\geq$-estimation is strict. For the termination hypotheses it suffices to show $\bigvee \Delta$, which is trivial in many cases.

Our new notion of *call-bounded* second-order procedures automates termination proofs for procedures that are defined by second-order recursion. Call-boundedness can be automatically detected for commonly used second-order procedures such as *map*, *every*, *foldl*, and *foldr*, for instance. Altogether this approach leads to automated termination proofs for a large number of procedures; see Chapters 6 and 7 for a comparison with related work and an experimental evaluation.

Chapter 5

Inductive Theorem Proving

According to Definition 2.84 (p. 59), a formula $\phi := \forall x : \tau.\ b$ over $\Sigma(P)$ for a terminating program P is true iff $eval_{P'}(b[q]) = true$ for all terminating programs $P' \supseteq P$, all grounding type substitutions $\theta \in GndSubst_{\Omega(P')}(\tau)$, and all values $q \in \mathbb{V}(P')_{\theta(\tau)}$. Since this definition quantifies over infinitely many programs P' and infinitely many values q_i, we need to find a way to prove $eval_{P'}(b[\vec{q}]) = true$ by finite means (i. e., within finite time and finite space) if we would like to show that formula ϕ is true.

A standard technique to prove universally quantified statements is *well-founded induction* (also called *Noetherian induction*): Let $\succ \subseteq S \times S$ be a well-founded relation on some set S. Then

$$\forall x \in S.\ \big(\forall x' \in S.\ x \succ x' \implies P[x']\big) \implies P[x] \tag{5.1}$$

implies

$$\forall x \in S.\ P[x]\ . \tag{5.2}$$

The implication "(5.1) $\implies$ (5.2)" is called an *induction axiom*. We call (5.1) the *induction formula* for (5.2). In order to prove induction formula (5.1), we need to prove $P[x]$ for an arbitrary $x \in S$. We may use that $P[x']$ holds for all $x' \in S$ with $x \succ x'$. The hypothesis that $P[x']$ holds for all $x' \in S$ with $x \succ x'$ is usually called the *induction hypothesis*.

Hence we can prove the truth of formula ϕ by choosing any well-founded relation $\succ$ on $\mathbb{V}(P')_{\theta(\tau)}$ and by showing

$$\begin{aligned}&\big(\forall q' \in \mathbb{V}(P')_{\theta(\tau)}.\ q \succ q' \implies eval_{P'}(b[q']) = true\big)\\ &\implies eval_{P'}(b[q]) = true\end{aligned} \tag{5.3}$$

for all terminating programs $P' \supseteq P$, all grounding type substitutions θ, and all values $q \in \mathbb{V}(P')_{\theta(\tau)}$.

Example 5.1. Suppose that we would like to show that formula

$$\forall n : \mathbb{N}.\ even(dbl(n))$$

is true. For an arbitrary extension[1] P' of the program P that contains the definitions of procedures *dbl* and *even* (see Figures 1.6 and 5.1) we define a relation $\succ$ on $\mathbb{V}(P')_{\mathbb{N}}$ by

$$\mathsf{n} \succ \mathsf{n}' \iff \mathsf{n} = {}^{+}(\mathsf{n}')\ . \tag{5.4}$$

By well-founded induction on n wrt. relation $\succ$ it suffices to show that the corresponding instance

$$\begin{aligned}&\big(\forall \mathsf{n}' \in \mathbb{V}(P')_{\mathbb{N}}.\ \mathsf{n} \succ \mathsf{n}' \implies eval_{P'}(even(dbl(\mathsf{n}'))) = true\big)\\ &\implies eval_{P'}(even(dbl(\mathsf{n}))) = true\end{aligned}$$

of (5.3) holds.

Case $\mathsf{n} = \mathsf{0}$: $eval_{P'}(even(dbl(\mathsf{0}))) = eval_{P'}(even(\mathsf{0})) = true$.

Case $\mathsf{n} \neq \mathsf{0}$: Then $\mathsf{n} = {}^{+}(\mathsf{n}')$ for some $\mathsf{n}' \in \mathbb{V}(P')_{\mathbb{N}}$ and we get:

$$\begin{aligned}&eval_{P'}(even(dbl({}^{+}(\mathsf{n}'))))\\ &= eval_{P'}(even({}^{+}({}^{+}(dbl(\mathsf{n}'))))) && \text{; by def. of } dbl\\ &= eval_{P'}(even(dbl(\mathsf{n}'))) && \text{; by def. of } even\\ &= true && \text{; because } \mathsf{n} \succ \mathsf{n}'\end{aligned} \tag{5.5}$$

In the last step we used the induction hypothesis. ♢

In the example, we performed a case analysis over n. Case $\mathsf{n} = \mathsf{0}$ is usually called the *base case* of the induction. The base case is given by those elements that are minimal wrt. relation $\succ$; since there is no $\mathsf{n}' \in \mathbb{V}(P')_{\mathbb{N}}$ with $\mathsf{0} \succ \mathsf{n}'$ in the example, $\mathsf{n} = \mathsf{0}$ is a base case. Case $\mathsf{n} \neq \mathsf{0}$ is the *step case* of the induction. In general, the step case considers those elements that are *not* minimal wrt. relation $\succ$.

Note that we always denoted the $\succ$-predecessor(s) of some x by a *primed term variable* x'. Hence we generally assume that each family $\mathcal{V}$ of term variables does not contain primed term variables and define the family $\mathcal{V}'$ of primed term variables by:

$$\mathcal{V}'_\tau := \big\{\, x' \;\big|\; x \in \mathcal{V}_\tau \big\}$$

[1] Actually, we could do without the extension of program P, because the formula does not contain variables of function type.

```
procedure even(n:ℕ) : bool <=
if ?0(n)
  then true
  else if ?0(⁻(n))
         then false
         else even(⁻(⁻(n)))
end    end
```

Figure 5.1: Procedure *even* that decides if a natural number is even

Organization of this chapter. In Section 5.1 we describe the general architecture for inductive proofs in ✓eriFun. In Section 5.2 we show how well-founded relations $\succ$ can be obtained from data structure definitions and from procedures that have been proved terminating. In the example above, $\succ$ is the well-founded relation that is obtained from the definition of data structure $\mathbb{N}$.

We consider the synthesis of induction formulas based on these well-founded relations in Section 5.3. The induction formulas are proved by so-called *symbolic evaluation*. Section 5.4 presents the extensions of ✓eriFun's calculus for symbolic evaluation that facilitate reasoning about λ-expressions and second-order procedures.

5.1 Inductive Proofs in ✓eriFun

The proof of an induction formula typically starts with a case analysis over the induction variable: If the induction variable denotes a value that is minimal wrt. the chosen well-founded relation, this case is a *base case*; otherwise it is a *step case*. In Example 5.1, the base case of the induction is characterized by literal $n = 0$ and the step case is characterized by literal $n \neq 0$. Sometimes more refined case analyses are useful that distinguish between several base cases or several step cases.

In ✓eriFun, the individual cases of a proof by induction are represented by *sequents* $\langle H, IH \Vdash goal\rangle$ [73, 89, 91, 95]. Each case is defined by a set H of *hypotheses*. Set IH contains the *induction hypotheses* of the case under consideration. Induction hypotheses may contain primed variables, which are considered as universally quantified. (Here the primed variables are those term variables that are *not* induction variables, see Section 5.3.) The term that is to be shown to evaluate to *true* is called the *goal term*.

Definition 5.2 (Sequents)**.** *A* sequent *for a terminating program P and a finite family $\mathcal{V}$ of term variables $x_1 : \tau_1, \ldots, x_n : \tau_n$ is a triple $\langle H, IH \Vdash goal\rangle$ of a set $H \in \mathcal{CL}(\Sigma(P), \mathcal{V})$ of* hypotheses*, a set $IH \in \mathcal{CL}(\Sigma(P), \mathcal{V} \cup \mathcal{V}')$ of* induction hypotheses*, and a* goal term *$goal \in \mathcal{T}(\Sigma(P), \mathcal{V})_{bool}$.*

Sequent $\langle H, IH \Vdash goal\rangle$ *is* true *iff for all terminating programs* $P' \supseteq P$, *all grounding type substitutions* $\theta \in GndSubst_{\Omega(P')}(\tau_1, \ldots, \tau_n)$, *and all values* $q_1, \ldots, q_n$ *with* $q_i \in \mathbb{V}(P')_{\theta(\tau_i)}$ *for all* $i = 1, \ldots, n$, *the conjunction of*

- $eval_{P'}(h[\vec{q}]) = true$ *for all* $h \in H$ *and*
- $eval_{P'}(ih[\vec{q}, \vec{q'}]) = true$ *for all* $ih \in IH$ *and all values* $q'_1, \ldots, q'_n$ *with* $q'_i \in \mathbb{V}(P')_{\theta(\tau_i)}$ *for all* $i = 1, \ldots, n$

entails $eval_{P'}(goal[\vec{q}]) = true$.

In ✓eriFun, formulas are proved by using a sequent calculus, called the *HPL-calculus* [73, 91, 95], where HPL abbreviates *Hypotheses, Programs, and Lemmas*. The set of sequents $\langle H, IH \Vdash goal\rangle$ defines the language of the HPL-calculus. Each proof rule of the HPL-calculus transforms a sequent into a finite set of sequents. A proof is represented by a finite *proof tree* whose nodes are labeled with sequents. For a formula $\forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ b$, the root node is given by the *initial sequent* $\langle \emptyset, \emptyset \Vdash b\rangle$. The labels of the child nodes are obtained by applying a proof rule to the label of the parent node. Each proof rule is *sound* in the sense that the truth of all child sequents entails the truth of the parent sequent. If each leaf of the proof tree is of the form $\langle \ldots, \ldots \Vdash true\rangle$, the proof tree is *closed*, and we thus have a proof of the formula.

In our context, the following proof rules of the HPL-calculus are of particular interest (for further proof rules see [73, 91, 95]):

Induction creates the base and step sequents for an initial sequent wrt. a well-founded relation $\mathcal{R}$:

$$\frac{\langle \emptyset, \emptyset \Vdash goal\rangle}{\langle H_1, IH_1 \Vdash goal\rangle, \ldots, \langle H_n, IH_n \Vdash goal\rangle}$$

In Section 5.3 we describe in detail how the base and step sequents $IndForm_{\mathcal{R}}(goal) = \{\langle H_1, IH_1 \Vdash goal\rangle, \ldots, \langle H_n, IH_n \Vdash goal\rangle\}$ for an initial sequent $\langle \emptyset, \emptyset \Vdash goal\rangle$ are computed.

Simplification applies an automated theorem prover, the so-called *symbolic evaluator* [73, 91, 95], to a sequent. Starting with the goal term *goal* of a sequent, the *symbolic evaluator* iteratively applies inference rules of the *evaluation calculus* to this term until a term $goal_\perp$ is obtained to which no further evaluation rule can be applied:

$$\frac{\langle H, IH \Vdash goal\rangle}{\langle H, IH \Vdash goal_\perp\rangle}$$

We look at the aspects of symbolic evaluation concerning λ-expressions and second-order procedures in Section 5.4.

Use Lemma applies an instance of a lemma

$$\texttt{lemma}\ \mathit{name} <= \forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ b$$

to a goal term *goal* at some term position $\pi \in Pos(goal)$. Term substitution σ instantiates the universally quantified variables $x_1, \ldots, x_n$:

$$\frac{\langle H, IH \Vdash goal\rangle}{\langle H, IH \Vdash goal[\pi \leftarrow if\{\sigma(b), goal|_\pi, true\}]\rangle}$$

Induction hypotheses from *IH* can be used in the same way by considering them as lemmas.

Apply Equation replaces a subterm of *goal* with an equal term. All lemmas and induction hypotheses are represented by (disjunctive) clauses. Thus a conditional equation $c_1 \wedge \ldots \wedge c_n \rightarrow l = r$ is represented by a clause $C = \{\neg c_1, \ldots, \neg c_n, l = r\}$, for instance.

For a clause C with $l = r \in C$, a term substitution σ that instantiates the universally quantified variables of C, a term position $\pi \in Pos(goal)$ with $\sigma(l) = goal|_\pi$, and $C' := C \setminus \{l = r\}$, *Apply Equation* is defined by

$$\frac{\langle H, IH \Vdash goal\rangle}{\langle H, IH \Vdash goal[\pi \leftarrow if\{\mathsf{NOR}(\sigma(C')), \sigma(r), goal|_\pi\}]\rangle}\ ,$$

where $\mathsf{NOR}(\sigma(C'))$ is a Boolean term that represents the conjunction of the negated literals in $\sigma(C')$.

✓eriFun's *Verify Tactic* builds a proof tree by heuristically applying some proof rules. A proof typically starts with *Induction* wrt. an induction axiom suggested by the system's *induction heuristic* [83, 85].[2] Then the tactic tries to close the proof tree by applying *Simplification* to the child nodes. It also employs *Use Lemma* and *Apply Equation* to use heuristically helpful induction hypotheses if they have not already been used by the previous *Simplification*.

Each lemma and each procedure of a program P has a certain *status* $\in$ {ignored, ready, terminating, verified} [73, 91, 95]. A lemma possesses status

ignored if it uses a procedure *proc* with a status different from verified (for instance, because termination of procedure *proc* has not been proved yet) or if a context hypothesis of the lemma has status different from verified (i. e., it is not yet confirmed that the context requirement of all function symbols occurring in the lemma are satisfied);

ready if all procedures that the lemma calls and all context hypotheses possess status verified, but the proof tree of the lemma is *not* closed;

[2] We did not need to modify the induction heuristic for our approach.

verified if all procedures that the lemma calls and all context hypotheses possess status verified and the proof tree of the lemma is closed.

A procedure *proc* possesses status

ignored if *proc* calls a procedure with a status different from verified or if no termination hypotheses have been generated for *proc* yet;

ready if all procedures that *proc* calls possess status verified and there exists some termination hypothesis for *proc* with status different from verified;

terminating if all procedures that *proc* calls possess status verified and there exists a (finite) set of termination hypotheses for *proc* with status verified and some context hypothesis possesses a status different from verified;

verified if all procedures that *proc* calls possess status verified and there exists a (finite) set of termination hypotheses for *proc* with status verified and all context hypotheses of *proc* possess status verified.

5.2 Representation of Relations

An infinite relation $\succ \subseteq \mathbb{N} \times \mathbb{N}$ cannot be written down by enumerating all pairs $(n, n') \in \succ$. Usually such a relation is defined by a *formula* that characterizes the elements of $\succ$. For instance, the (meta-level) formula

$$\forall n, n' : \mathbb{N}.\ (n, n') \in \succ \iff n \neq 0 \wedge n' = n - 1 \tag{5.6}$$

defines a relation $\succ$ on $\mathbb{N}$ with $n \succ n'$ iff $n = n' + 1$.

Since we are interested in proving $\mathcal{L}$-formulas by well-founded induction, we need a representation of well-founded relations on $\mathbb{V}(P)$. We can use the same idea as for the representation of relation $\succ$ in (5.6). For example, the *atomic relation representation*

$$A_1[n, n'] \ :\Longleftrightarrow\ ?^+(n) \wedge n' = {}^-(n) \tag{5.7}$$

defines a relation $\succ_{A_1,n}$ on $\mathbb{V}(P)_{\mathbb{N}}$ with $\mathsf{n} \succ_{A_1,n} \mathsf{n'}$ iff $\mathsf{n} = {}^+(\mathsf{n'})$. This is the same relation that we used in Example 5.1 (p. 131).

We call $?^+(n)$ in (5.7) a *domain literal* of A_1. The domain literal contains only unprimed term variables and restricts the values that can have predecessors wrt. this relation.

The so-called *range predicate* in general contains both primed an unprimed variables. In (5.7), the range predicate $n' = {}^-(n)$ relates n to its predecessor n'. Range predicates need not be equations $x' = f(x)$. They can also use quantification procedures. For instance, the following atomic

relation representation A_2 represents the usual *direct subterm* relation on terms:

$$A_2[t, t'] \;:\Longleftrightarrow\; ?\mathit{apply}(t) \wedge \mathit{exists}.\mathit{list}(\lambda s : \mathit{term}[@\,V, @F].\, t' = s, \mathit{args}(t)) \quad (5.8)$$

A term t' is a direct subterm of t iff $?\mathit{apply}(t)$ and $t' = s$ for some $s \in \mathit{args}(t)$.[3]

Compared with the notation in (5.4), the use of domain literals makes it easier to find out which elements are minimal wrt. the relation (and thus form the base case of the induction) and which elements are not minimal (and thus form the step case of the induction). To mark base cases even more clearly, we also allow the constant *false* as range predicate. For instance, we can complement the atomic relation representation A_1 by

$$A_1'[n, n'] \;:\Longleftrightarrow\; ?\mathsf{0}(n) \wedge \mathit{false}\ . \quad (5.9)$$

Although $\mathsf{n} \succ_{A_1', n} \mathsf{n}'$ for no $\mathsf{n}, \mathsf{n}' \in \mathbb{V}(P)_{\mathbb{N}}$, such atomic relation representations that represent empty relations are useful to synthesize appropriate base cases for an induction wrt. a relation, see Section 5.3.

Definition 5.3 (Range predicates)**.** *The set $\mathcal{RP}(\Sigma, \mathcal{V}) \subset \mathcal{T}(\Sigma, \mathcal{V} \cup \mathcal{V}')_{bool}$ of* range predicates *over a term signature Σ and a family $\mathcal{V}$ of term variables is defined by $R \in \mathcal{RP}(\Sigma, \mathcal{V})$ iff either*

1. *$R = \mathit{false}$,*
2. *$R = \bigwedge_{i=1}^{n} x_i' = t_i$ for some $n \geq 1$, $x_i' \in \mathcal{V}'$, and $t_i \in \mathcal{T}(\Sigma, \mathcal{V})$ for all $i = 1, \ldots, n$, or*
3. *$R = \mathit{exists}(\lambda y_1 : \tau_1, \ldots, y_m : \tau_m.\, \bigwedge D' \wedge R',\ t_1, \ldots, t_k)$ for some* local domain clause *$D' \in \mathcal{CL}(\Sigma, \mathcal{V} \cup \{y_1, \ldots, y_m\})$, some* local range predicate *$R' \in \mathcal{RP}(\Sigma, \mathcal{V} \cup \{y_1, \ldots, y_m\})$, and some procedure $\mathit{exists} \in \Sigma^{\mathsf{ex}}$.*

Definition 5.4 (Relation representations)**.** *Let Σ be a term signature and let $\mathcal{V}$ be a finite family of term variables. An* atomic relation representation *over Σ and $\mathcal{V}$ is a Boolean term $A \in \mathcal{T}(\Sigma, \mathcal{V} \cup \mathcal{V}')_{bool}$ of the form*

$$A = \bigwedge D \wedge R$$

for a domain clause *$D \in \mathcal{CL}(\Sigma, \mathcal{V})$ and a* range predicate *$R \in \mathcal{RP}(\Sigma, \mathcal{V})$. A* composed relation representation *(or* relation representation *for short) is a finite disjunction $\mathcal{R} = A_1 \vee \ldots \vee A_k$ of atomic relation representations. $\mathcal{REL}(\Sigma, \mathcal{V})$ denotes the set of all relation representations over Σ and $\mathcal{V}$.*

[3] Admittedly, relation representation (5.8) is not as easy to read as the equivalent representation $?\mathit{apply}(t) \wedge t' \in \mathit{args}(t)$. However, relation representations are just an *internal* representation of relations within a theorem prover so a system user does not need to investigate them. The formulation as in (5.8) is beneficial wrt. the synthesis of induction axioms, see Sections 5.3 and 7.2.2.

Definition 5.5 (Semantics of a relation representation)**.** *Let $\mathcal{R} = A_1 \vee \ldots \vee A_k$ be a relation representation over $\Sigma(P)$ and $\mathcal{V}$ for some program P. Furthermore, let $\hat{\mathcal{V}} \supseteq \mathcal{V}$ be a family of term variables and let $x^* := x_1 \ldots x_n \in \hat{\mathcal{V}}^*$ be a sequence of n distinct term variables $x_i : \tau_i$ such that $\mathcal{V} \subseteq \{x_1, \ldots, x_n\}$.*

For an atomic relation representation $A = \bigwedge D \wedge R$ of $\mathcal{R}$ and a grounding type substitution $\theta \in GndSubst_{\Omega(P)}(\tau_1, \ldots, \tau_n)$, the relation $\succ_{A,\theta,x^*}$ on $\mathbb{V}(P)_{\theta(\tau_1)} \times \ldots \times \mathbb{V}(P)_{\theta(\tau_n)}$ is defined by

$$(q_1, \ldots, q_n) \succ_{A,\theta,x^*} (q'_1, \ldots, q'_n) \;:\Longleftrightarrow\; eval_P(\sigma(A)) = true$$

where $\sigma := \{x_1/q_1, \ldots, x_n/q_n, x'_1/q'_1, \ldots, x'_n/q'_n\}$. The relation $\succ_{\mathcal{R},\theta,x^*}$ on $\mathbb{V}(P)_{\theta(\tau_1)} \times \ldots \times \mathbb{V}(P)_{\theta(\tau_n)}$ is defined by $\succ_{\mathcal{R},\theta,x^*} := \succ_{A_1,\theta,x^*} \cup \ldots \cup \succ_{A_k,\theta,x^*}$. Relation representation $\mathcal{R}$ is well-founded *iff $\succ_{\mathcal{R},\theta,x^*}$ is well-founded for some $x^* \in \mathcal{V}^*$ and all $\theta \in GndSubst_{\Omega(P)}(\tau_1, \ldots, \tau_n)$.*

Obviously, it is decidable if $(q_1, \ldots, q_n) \succ_{\mathcal{R},\theta,x^*} (q'_1, \ldots, q'_n)$ for a relation representation $\mathcal{R}$, a grounding type substitution θ, and values $q_1, \ldots, q_n$ and $q'_1, \ldots, q'_n$. Consequently, not all relations on values can be described by a relation representation, as there are undecidable relations. However, such undecidable relations are practically irrelevant in our setting and the relations that we investigate in the following subsections are decidable.

Before getting to these concrete relation representations, we introduce the notion of a *case complete* relation representation:

Definition 5.6 (Case complete relation representations)**.** *Let $\mathcal{R} = A_1 \vee \ldots \vee A_k$ be a relation representation over $\Sigma(P)$ and $\mathcal{V} = \{x_1, \ldots, x_n\}$ for some program P. Let $x_i : \tau_i$ for each $x_i \in \mathcal{V}$.*

Relation representation $\mathcal{R}$ is case complete *iff for all type substitutions $\theta \in GndSubst_{\Omega(P)}(\tau_1, \ldots, \tau_n)$ and all values $q_1, \ldots, q_n$ with $q_i \in \mathbb{V}(P)_{\theta(\tau_i)}$ for $i = 1, \ldots, n$ there is some $A_j = \bigwedge D \wedge R$ (where $j \in \{1, \ldots, k\}$) such that $eval_P(d[q_1, \ldots, q_n]) = true$ for each $d \in D$.*

Example 5.7. $\mathcal{R}[n, n'] \;:\Longleftrightarrow\; A_1[n, n'] \vee A'_1[n, n']$ is a case complete relation representation with $\mathsf{n} \succ_{\mathcal{R},n} \mathsf{n}'$ iff $\mathsf{n} = {}^+(\mathsf{n}')$ for $\mathsf{n}, \mathsf{n}' \in \mathbb{V}(P)_{\mathbb{N}}$. ♢

Remark 5.8. Definitions 5.3, 5.4, and 5.5 generalize the concept of *relation descriptions* in [83, 85, 89]. There the predecessors wrt. a relation are represented by *range substitutions* instead of *range predicates*. A range substitution is a (partial) term substitution $\{x_1/t_1, \ldots, x_n/t_n\}$.[4] The straightforward translation of such a range substitution into a range predicate is $x'_1 \mathsf{=} t_1 \wedge \ldots \wedge x'_n \mathsf{=} t_n$.

[4] A "partial" term substitution $\{x_1/t_1, \ldots, x_n/t_n\}$ differs from a usual term substitution in that it can only be applied to a term t with $\mathcal{V}_t(t) \subseteq \{x_1, \ldots, x_n\}$. The partial term substitution $\{x_1/f(a), x_2/x_2\}$ is different from the partial term substitution $\{x_1/f(a)\}$, because the first partial term substitution can be applied to term $g(x_1, x_2)$, whereas the second one is not applicable.

For example, consider the atomic relation representation A_1 from (5.7). In [85, 89], range predicate $n' = {}^{-}(n)$ is represented by the term substitution $\delta := \{n/{}^{-}(n)\}$.

However, it is impossible to finitely enumerate the predecessors in a relation such as the *direct subterm* relation, cf. A_2 in (5.8). We need to represent such relations in order to obtain the usual induction axiom for structural induction on data structure $term[@V, @F]$ for terms. Using relation *descriptions*, one would have to write something like

$$\{t/hd(args(t)),\ t/hd(tl(args(t))),\ t/hd(tl(tl(args(t)))),\ \ldots\}\ .$$

Our quantification procedures from Chapter 3 allow us to capture these arbitrary many, but finitely many predecessors.

It is still useful to think of a range predicate $x'_1 = t_1 \wedge \ldots \wedge x'_n = t_n$ as a term substitution $\{x_1/t_1, \ldots, x_n/t_n\}$, because we interpret such conjunctions of equations as term substitutions when we synthesize induction axioms in Section 5.3.

5.2.1 Well-Founded Relations from Data Structures

For a data structure definition

$$\begin{array}{l}\texttt{structure}\ str[@A_1, \ldots, @A_k] <= \\ \quad \ldots, \\ \quad cons(sel_1 : \tau_1, \ldots, sel_n : \tau_n), \\ \quad \ldots\end{array} \tag{5.10}$$

of a program P (cf. Definition 2.31 on p. 30) one can uniformly synthesize a relation representation for proofs by *structural induction* on a variable of type $str[\tau_1, \ldots, \tau_k]$.

The domain literals of such a relation representation are of the form $?cons(x)$ as in (5.7) and (5.8). To synthesize range predicates like $n' = {}^{-}(n)$ and $exists.list(\lambda s : term[@V, @F].\, t' = s, args(t))$, we use the following construction:

For a base type $\tau = str[\tau_1, \ldots, \tau_k]$, a term $t \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$, a type position $\pi \in Pos(\tau)$, and a term $t' \in \mathcal{T}(\Sigma, \mathcal{V})_{\tau|_\pi}$, we define

$$R_\tau(t, t', \pi) := \begin{cases} t' = t & \text{if } \pi = \epsilon \\ exists.str_h(\lambda y : \tau_h.\, R_{\tau_h}(y, t', \pi'),\ t) & \text{if } \pi = h\pi'\,. \end{cases} \tag{5.11}$$

Intuitively, $R_\tau(t, t', \pi)$ yields a Boolean term that evaluates to *true* iff $t' \in Itm_\tau(t, \pi)$, cf. Definition 2.56 (p. 42). We use these terms $R_\tau(t, t', \pi)$ as range predicates in the relation representation of a data structure.

Definition 5.9 (Relation representation of a data structure)**.** *For a data structure definition of the form (5.10), let $\mathcal{V}$ be a family of term variables that contains just $x : str[@A_1, \ldots, @A_k]$.*

For each reflexive str-constructor cons and each $(j, \pi) \in Occ_{str}(cons)$, the atomic relation representation $A_{cons,j,\pi}$ is defined by

$$A_{cons,j,\pi}[x, x'] \;:\Longleftrightarrow\; ?cons(x) \wedge R_{\tau_j}(sel_j(x), x', \pi)\,.$$

The relation representation $\mathcal{R}_{str}$ of *str is defined by*

$$\begin{aligned}\mathcal{R}_{str}[x, x'] \;:\Longleftrightarrow\;& \\ &\bigvee \left\{ A_{cons,j,\pi}[x, x'] \;\middle|\; cons \in \mathcal{C}^{\mathsf{refl}}_{str} \text{ and } (j, \pi) \in Occ_{str}(cons) \right\} \vee \\ &\bigvee \left\{ ?cons(x) \wedge false \;\middle|\; cons \in \mathcal{C}^{\mathsf{irr}}_{str} \right\}\,.\end{aligned}$$

We get the following relation representations for the data structure definitions of Figure 2.1 (p. 31):

Example 5.10. Type constructor $\mathbb{N}$ has one irreflexive data constructor $\mathsf{0}$ and one reflexive data constructor ${}^{+}(\ldots)$ with $Occ_{\mathbb{N}}({}^{+}) = \{(1, \epsilon)\}$. Thus

$$\mathcal{R}_{\mathbb{N}}[x, x'] \;:\Longleftrightarrow\; [?\mathsf{0}(x) \wedge false] \vee [?{}^{+}(x) \wedge x' = {}^{-}(x)]\,. \qquad \diamondsuit$$

Example 5.11. Type constructor *list* has one irreflexive data constructor ε and one reflexive data constructor :: with $Occ_{list}(::) = \{(2, \epsilon)\}$. Thus

$$\mathcal{R}_{list}[x, x'] \;:\Longleftrightarrow\; [?\varepsilon(x) \wedge false] \vee [?::(x) \wedge x' = tl(x)]\,. \qquad \diamondsuit$$

Example 5.12. Type constructor *pair* has no reflexive data constructor, so

$$\mathcal{R}_{pair}[x, x'] \;:\Longleftrightarrow\; ?\bullet(x) \wedge false\,. \qquad \diamondsuit$$

Example 5.13. Type constructor *term* has one reflexive data constructor *apply* with $Occ_{term}(apply) = \{(2, \mathbf{1})\}$. Thus

$$\begin{aligned}&A_{apply,2,\mathbf{1}}[x, x'] \\ &\quad :\Longleftrightarrow ?apply(x) \wedge R_{list[term[@V, @F]]}(args(x), x', \mathbf{1}) \\ &\quad \Longleftrightarrow ?apply(x) \wedge exists.list(\lambda y : term[@V, @F].\, x' = y,\; args(x))\,.\end{aligned}$$

The other *term*-constructor *var* is irreflexive, so

$$\begin{aligned}\mathcal{R}_{term}[x, x'] \;:\Longleftrightarrow\;& \\ &[?var(x) \wedge false] \vee \\ &[?apply(x) \wedge exists.list(\lambda y : term[@V, @F].\, x' = y,\; args(x))]\,.\end{aligned}$$

$\mathcal{R}_{term}$ represents the *direct subterm* relation on terms. $\diamondsuit$

Example 5.14. Type constructor *mylist* (cf. Figure 5.2) has one reflexive data constructor *add* with $Occ_{mylist}(add) = \{(1, \mathbf{2})\}$. Thus

$$\begin{aligned} & A_{add,1,\mathbf{2}}[x, x'] \\ & \quad :\Longleftrightarrow ?add(x) \wedge R_{pair[@A,mylist[@A]]}(entry(x), x', \mathbf{2}) \\ & \quad \Longleftrightarrow ?add(x) \wedge exists.pair_2(\lambda y : mylist[@A].\, x' = y,\ entry(x)) \\ & \quad \Longleftrightarrow ?add(x) \wedge x' = snd(entry(x)) \end{aligned}$$

by replacing $exists.pair_2(\ldots)$ with the instantiated body of $exists.pair_2$, because $exists.pair_2$ is *not* defined recursively (or rather, $forall.pair_2$ is not defined recursively). Together with the irreflexive *mylist*-constructor *empty*, we get the relation representation

$$\begin{aligned} \mathcal{R}_{mylist}[x, x'] \ :\Longleftrightarrow\ & [?empty(x) \wedge false] \vee \\ & [?add(x) \wedge x' = snd(entry(x))] \end{aligned}$$

as expected. ♢

Example 5.15. Type constructor *bin.tree* (cf. Figure 5.2) has one irreflexive data constructor *tip* and one reflexive data constructor *node*. Since $Occ_{bin.tree}(node) = \{(1, \epsilon), (3, \epsilon)\}$, we get

$$\begin{aligned} \mathcal{R}_{bin.tree}[x, x'] \ :\Longleftrightarrow\ & [?tip(x) \wedge false] \vee \\ & [?node(x) \wedge x' = left(x)] \vee \\ & [?node(x) \wedge x' = right(x)]\,. \end{aligned}$$

The left and the right subtree of an inner node are the predecessors of a binary tree wrt. this relation representation. ♢

Example 5.16. Type constructor *tree* (cf. Figure 4.10 on p. 124) has one reflexive data constructor *branch* with $Occ_{tree}(branch) = \{(1, \mathbf{1})\}$. Thus

$$\begin{aligned} & A_{branch,1,\mathbf{1}}[x, x'] \\ & \quad :\Longleftrightarrow ?branch(x) \wedge R_{list[tree[@A]]}(children(x), x', \mathbf{1}) \\ & \quad \Longleftrightarrow ?branch(x) \wedge exists.list(\lambda y : tree[@A].\, x' = y,\ children(x))\,. \end{aligned}$$

Together with the irreflexive *tree*-constructor *leaf* we get

$$\begin{aligned} \mathcal{R}_{tree}[x, x'] \ :\Longleftrightarrow\ & \\ & [?leaf(x) \wedge false] \vee \\ & [?branch(x) \wedge exists.list(\lambda y : tree[@A].\, x' = y,\ children(x))]\,. \end{aligned}$$

$\mathcal{R}_{tree}$ represents the *direct subtree* relation on variadic trees. ♢

```
structure mylist[@A] <=
  empty,
  add(entry : pair[@A, mylist[@A]])

structure bin.tree[@A] <=
  tip,
  node(left : bin.tree[@A], key : @A, right : bin.tree[@A])
```

Figure 5.2: Data structure definitions $mylist[@A]$ and $bin.tree[@A]$

Theorem 5.17. *Relation representation $\mathcal{R}_{str}$ is well-founded and case complete for each data structure $str[@A_1, \ldots, @A_k]$.*

Proof. First we show that $eval_P(R_\tau(q, q', \pi)) = true$ entails $q' \leq_\mathcal{T} q$ for all ground base types τ, $\pi \in Pos(\tau)$, $q \in \mathbb{V}(P)_\tau$, and $q' \in \mathbb{V}(P)_{\tau|_\pi}$. We show this by structural induction on q.

If $\pi = \epsilon$, then $eval_P(q' = q) = true$, so $q' = q \leq_\mathcal{T} q$.

If $\pi = h\pi'$, then $eval_P(exists.str_h(\lambda y : \tau_h.\, R_{\tau_h}(y, q', \pi'),\ q)) = true$. By Lemma 3.11 (p. 77), this is equivalent to $eval_P(R_{\tau_h}(q'', q', \pi')) = true$ for some $q'' \in Itm_\tau(q, h)$. By the induction hypothesis, $q' \leq_\mathcal{T} q''$. Since $q'' <_\mathcal{T} q$, $q' \leq_\mathcal{T} q$.

Now we show that $\succ_{\mathcal{R}_{str},\theta,x}$ is well-founded for each type substitution $\theta \in GndSubst_{\Omega(P)}(@A_1, \ldots, @A_k)$. Let $q, q' \in \mathbb{V}(P)_{\theta(\tau)}$ with $q \succ_{\mathcal{R}_{str},\theta,x} q'$, where $\tau := str[@A_1, \ldots, @A_k]$. Then $q \succ_{A_{cons,j,\pi},\theta,x} q'$ for some $cons \in \mathcal{C}_{str}$ and some $(j, \pi) \in Occ_{str}(cons)$. Thus (†) $eval_P(?cons(q)) = true$ and (‡) $eval_P(R_{\tau_j}(sel_j(q), q', \pi)) = true$.

From (†) we conclude $q = cons(q_1, \ldots, q_n)$ for some $q_j \in \mathbb{V}(P)_{\theta(\tau_j)}$. From (‡) we conclude $q' \leq_\mathcal{T} q_j$. Since $q_j <_\mathcal{T} q$, we get $q' <_\mathcal{T} q$. Since relation $<_\mathcal{T}$ is well-founded, so is $\succ_{\mathcal{R}_{str},\theta,x}$.

$\mathcal{R}_{str}$ is case complete, because for each str-constructor $cons$, $\mathcal{R}_{str}$ contains an atomic relation representation with domain clause $\{?cons(x)\}$. □

5.2.2 Well-Founded Relations from Terminating Procedures

For a terminating procedure $proc : \tau_1 \times \ldots \times \tau_n \to \tau$ with parameters $x_1 : \tau_1, \ldots, x_n : \tau_n$, the recursive call relation $\succ^\theta_{proc}$ is well-founded for each grounding type substitution $\theta \in GndSubst_{\Omega(P)}(\tau_1, \ldots, \tau_n)$ (cf. Lemma 2.83 on p. 58). For procedure *even* (cf. Figure 5.1 on p. 133), we can represent relation $\succ_{even}$ by the relation representation

$$\begin{aligned}\mathcal{R}_{even}[n, n'] \ :\Longleftrightarrow\ & [?\mathtt{0}(n) \wedge false] \vee \\ & [\neg\, ?\mathtt{0}(n) \wedge ?\mathtt{0}(^-(n)) \wedge false] \vee \\ & [\neg\, ?\mathtt{0}(n) \wedge \neg\, ?\mathtt{0}(^-(n)) \wedge n' = {}^-({}^-(n))]\,.\end{aligned} \tag{5.12}$$

The base cases of an induction wrt. relation $\succ_{\mathcal{R}_{even},n}$ are $\mathsf{n} = \mathsf{0}$ and $\mathsf{n} = {}^{+}(\mathsf{0})$. This corresponds to the base cases of the recursive definition of procedure *even*. The step case $\mathsf{n} = {}^{+}({}^{+}(\mathsf{n}'))$ for some $\mathsf{n}' \in \mathbb{V}(P)_{\mathbb{N}}$ of the induction corresponds to the recursive call $even({}^{-}({}^{-}(n)))$.

The idea is to construct an atomic relation representation $A(B^{\mathsf{rel}}_{proc}, \pi)$ for each term position $\pi \in Pos(B^{\mathsf{rel}}_{proc})$ that either denotes a base case or a recursive call of *proc*. A recursive call may be a *direct* or an *indirect* recursive call. Therefore we consider three cases and generally define an atomic relation representation $A(t, \pi)[\vec{x}, \vec{x'}]$ for a normalized *let*-free term $t \in \mathcal{T}(\Sigma(P), \mathcal{V})$ and a term position $\pi \in \Pi^{base}_{proc}(t) \cup \Pi^{rec}_{proc}(t)$. Intuitively, $A(t, \pi)[\vec{x}, \vec{x'}]$ yields a Boolean term that evaluates to *true* iff evaluation of t requires the evaluation of a call of procedure *proc* at term position π with arguments $\vec{x'}$.

1. If $\pi \in \Pi^{base}_{proc}(t)$, then no call of *proc* needs to be evaluated:

$$A(t, \pi)[\vec{x}, \vec{x'}] \;:\Longleftrightarrow\; \bigwedge COND(t, \pi) \wedge \mathit{false} \tag{5.13}$$

2. If $\pi \in \Pi^{rec}_{proc}(t) \cap TLPos(t)$, then term position π denotes a *direct* recursive call $t|_\pi = proc(t_1, \ldots, t_n)$. This recursive call is evaluated iff the conditions of the call context $COND(t, \pi)$ are satisfied:

$$A(t, \pi)[\vec{x}, \vec{x'}] \;:\Longleftrightarrow\; \bigwedge COND(t, \pi) \wedge x'_1 = t_1 \wedge \ldots \wedge x'_n = t_n \tag{5.14}$$

3. If $\pi \in \Pi^{rec}_{proc}(t) \setminus TLPos(t)$, then term position π denotes an *indirect* recursive call and procedure *proc* is defined by second-order recursion. Hence there is a minimal prefix $\pi' \in TLPos(t)$ of π with $t|_{\pi'} = h(\lambda\vec{y}.\, t'',\, t')$ for a second-order procedure h; i. e., $\pi = \pi'\mathbf{10}\pi''$ for some $\pi'' \in Pos(t'')$ (cf. the construction of termination hypotheses for this case described in Section 4.1).

   The call of the second-order procedure h is evaluated iff the conditions of the call context $COND(t, \pi')$ are satisfied. Function $\lambda\vec{y}.\, t''$ is called by h iff $exists.h(\lambda\vec{y}.\, true,\; \lambda\vec{y}.\, t'',\; t')$ yields *true*. The indirect call of procedure *proc* in term t'' at position π'' is called with arguments $\vec{x'}$ iff $A(t'', \pi'')[\vec{x}, \vec{x'}]$ yields true. Thus we define for indirect recursive calls:

$$\begin{aligned} A(t, \pi)[\vec{x}, \vec{x'}] \;:\Longleftrightarrow\; & \bigwedge COND(t, \pi') \wedge \\ & exists.h(\lambda\vec{y}.\, A(t'', \pi'')[\vec{x}, \vec{x'}],\; \lambda\vec{y}.\, t'',\; t') \end{aligned} \tag{5.15}$$

This leads to the following definition of the relation representation of a procedure *proc*:

$$
\begin{array}{l}
\texttt{procedure}\ \mathit{varcount}(t : \mathit{term}[@V, @F]) : \mathbb{N} <= \\
\mathit{case}\ t\ \mathit{of} \\
\quad \mathit{var}\ \ : 1, \\
\quad \mathit{apply} : \mathit{foldl}(+, \mathtt{0}, \mathit{map}(\mathit{varcount}, \mathit{args}(t))) \\
\mathit{end}
\end{array}
$$

Figure 5.3: Counting the variables in a term using second-order recursion

Definition 5.18 (Relation representation of a procedure)**.** *Let $\mathcal{V}$ be the family of the formal parameters of a procedure*

$$\texttt{procedure}\ \mathit{proc}(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau <= B^{\mathsf{rel}}_{\mathit{proc}}$$

with let-free body $B^{\mathsf{rel}}_{\mathit{proc}}$. Relation representation $\mathcal{R}_{\mathit{proc}}$ *is defined by*

$$\mathcal{R}_{\mathit{proc}}[\vec{x}, \vec{x'}] \ :\Longleftrightarrow \bigvee \left\{ A(B^{\mathsf{rel}}_{\mathit{proc}}, \pi)[\vec{x}, \vec{x'}] \ \middle|\ \pi \in \Pi^{\mathit{base}}_{\mathit{proc}}(B^{\mathsf{rel}}_{\mathit{proc}}) \cup \Pi^{\mathit{rec}}_{\mathit{proc}}(B^{\mathsf{rel}}_{\mathit{proc}}) \right\}.$$

Example 5.19. Procedure *varcount* in Figure 5.3 computes the number of subterms of a term that are a variable. The relation representation of procedure *varcount* is given by

$$
\begin{array}{l}
\mathcal{R}_{\mathit{varcount}}[t, t'] \ :\Longleftrightarrow \\
\qquad [?\mathit{var}(t) \wedge \mathit{false}] \vee \\
\qquad [?\mathit{apply}(t) \wedge \mathit{exists}.\mathit{map}(\lambda s : \mathit{term}[@V, @F].\, t' = s, \\
\qquad\qquad\qquad\qquad \lambda s : \mathit{term}[@V, @F].\, \mathit{varcount}(s), \\
\qquad\qquad\qquad\qquad \mathit{args}(t))] .
\end{array}
$$

Thus $\mathsf{t} \succ_{\mathcal{R}_{\mathit{varcount}}, \theta, t} \mathsf{t'}$ iff $\mathsf{t} = \mathit{apply}(\mathsf{f}, \mathsf{t}_1 :: \ldots :: \mathsf{t}_n :: \varepsilon)$ for some $\mathsf{f} \in \mathbb{V}(P)_{\theta(@F)}$ and some $\mathsf{t}_1, \ldots, \mathsf{t}_n \in \mathbb{V}(P)_{\theta(\mathit{term}[@V, @F])}$ such that $\mathsf{t'} = \mathsf{t}_i$ for some $i = 1, \ldots, n$ (see Example 3.13 on p. 78 for an explanation of the semantics of procedure *exists.map*). ♢

Example 5.20. The relation representation of procedure *groundterm* (cf. Figure 1.5 on p. 9) is given by

$$
\begin{array}{l}
\mathcal{R}_{\mathit{groundterm}}[t, t'] \ :\Longleftrightarrow \\
\qquad [?\mathit{var}(t) \wedge \mathit{false}] \vee \\
\qquad [?\mathit{apply}(t) \wedge \mathit{exists}.\mathit{every}(\lambda s : \mathit{term}[@V, @F].\, t' = s, \\
\qquad\qquad\qquad\qquad \lambda s : \mathit{term}[@V, @F].\, \mathit{groundterm}(s), \\
\qquad\qquad\qquad\qquad \mathit{args}(t))] .
\end{array}
$$

Thus $\mathsf{t} \succ_{\mathcal{R}_{groundterm},\theta,t} \mathsf{t}'$ iff $\mathsf{t} = \mathit{apply}(\mathsf{f}, \mathsf{t}_1 :: \ldots :: \mathsf{t}_n :: \varepsilon)$ for some values $\mathsf{f} \in \mathbb{V}(P)_{\theta(@F)}$ and $\mathsf{t}_1, \ldots, \mathsf{t}_n \in \mathbb{V}(P)_{\theta(term[@V,@F])}$ such that there exists some $\nu \in \{1, \ldots, n\}$ with $\mathsf{t}' = \mathsf{t}_\nu$ and $\mathit{eval}_P(\mathit{groundterm}(\mathsf{t}_i)) = \mathit{true}$ for all $i < \nu$ (see Example 3.13 on p. 78 for an explanation of the semantics of procedure *exists.every*). ♢

A range predicate can involve equations $f' = t$ for a first-order variable f, which are—strictly speaking—syntactically wrong according to our definition of terms in Section 2.1. Recall that we wanted to avoid such equations, because equality of functions is undecidable in general. However, for relation representations we can relax this restriction for the following reasons:

1. Relation representations are just an internal representation of relations. They are only used by the theorem prover to synthesize induction axioms (see Section 5.3). There we consider $f' = t$ as a substitution $\{f'/t\}$, which is syntactically correct again.

2. The semantics of an equation $f' = t$ is that f' and t need to evaluate to *syntactically identical* λ-expressions. While it probably seems counterintuitive to a user that $t_1 := \lambda x.\, {}^+(x)$ and $t_2 := \lambda x.\, 1 + x$ are regarded as unequal, this just means for the semantics of a relation representation that one of these terms may be a predecessor of f', whereas the other term is no predecessor of f'. This makes sense, because the recursive call in a procedure *either* uses t_1 as argument *or* t_2.

Example 5.21. The relation representation for procedure *map* (cf. Figure 1.3 on p. 6) is

$$\begin{aligned}\mathcal{R}_{map}[f, k, f', k'] \;:\Longleftrightarrow\; & [?\varepsilon(k) \wedge \mathit{false}] \vee \\ & [\neg\, ?\varepsilon(k) \;\wedge\; f' = \lambda y.\, f(y) \;\wedge\; k' = \mathit{tl}(k)]\,.\end{aligned}$$

Thus $(\mathsf{f}, \mathsf{k}) \succ_{\mathcal{R}_{map},\theta,fk} (\mathsf{f}', \mathsf{k}')$ iff $\mathsf{k} = \mathsf{x} :: \mathsf{k}'$ for some $\mathsf{x} \in \mathbb{V}(P)_{\theta(@A)}$ and $\mathsf{f}' = \lambda y.\, \mathsf{f}(y)$. We will drop the unnecessary equation $f' = \lambda y.\, f(y)$ in Example 5.23 below. ♢

Theorem 5.22. *Relation representation $\mathcal{R}_{proc}$ is well-founded and case complete for each procedure proc of a terminating program P.*

Proof. Let procedure *proc* be defined by

$$\texttt{procedure}\ \mathit{proc}(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \mathrel{<=} B^{\mathsf{rel}}_{proc}\,.$$

We show that $\succ_{\mathcal{R}_{proc},\theta,x^*}$ is equal to the recursive call relation $\succ^{\theta}_{proc}$ from Definition 2.78 (p. 56) for $x^* := x_1 \ldots x_n$ and each grounding type substitution $\theta \in \mathit{GndSubst}_{\Omega(P)}(\tau_1, \ldots, \tau_n)$. The recursive call relation is well-founded by Lemma 2.83 (p. 58).

For direct recursive calls $f(q_1, \ldots, q_n) \rhd f(q'_1, \ldots, q'_n)$, relations $\succ^{\theta}_{proc}$ and $\succ_{\mathcal{R}_{proc},\theta,x^*}$ coincide, because both are defined via $COND(t, \pi)$.

For indirect recursive calls the equality follows from Lemma 3.12 (p. 77) by induction on the length m of the sequence

$$f(q_1, \ldots, q_n) \rhd h_1(\ldots) \rhd \ldots \rhd h_m(\ldots) \rhd f(q'_1, \ldots, q'_n)\,.$$

$\mathcal{R}_{proc}$ is case complete by construction: Either a recursive call needs to be evaluated or no recursive call needs to be evaluated (i. e., we get into a base case). For each such case there is an atomic relation representation. □

5.2.3 Optimization of Relation Representations

The relation representations of procedures according to Definition 5.18 are often suboptimal, because the corresponding induction axioms are overly specific. We start with an overview of the existing optimization techniques from [85, 89] and then show how relation representations of procedures with *second-order recursion* can be optimized.

Relation representations $\mathcal{R}$ are optimized by removing unnecessary details from the relation representation, which is called *generalization*. Semantically, a generalized relation representation $\mathcal{R}'$ *subsumes* relation representation $\mathcal{R}$ in the sense that $\succ_{\mathcal{R}',\theta,x^*} \supset \succ_{\mathcal{R},\theta,x^*}$. If relation representation $\mathcal{R}'$ is well-founded, then well-founded induction wrt. $\mathcal{R}'$ instead of well-founded induction wrt. $\mathcal{R}$ offers the following advantages:

- A base case turns into a step case if a $\succ_{\mathcal{R},\theta,x^*}$-minimal tuple $(q_1, \ldots, q_n)$ of values has a $\succ_{\mathcal{R}',\theta,x^*}$-predecessor. Hence this case can be proved with the additional support of an induction hypothesis.
- A step case gets stronger induction hypotheses if some tuple $(q_1, \ldots, q_n)$ of values has more $\succ_{\mathcal{R}',\theta,x^*}$-predecessors than $\succ_{\mathcal{R},\theta,x^*}$-predecessors. This generally makes it easier to prove the step case.

For instance, consider the relation representation of procedure "+" (cf. Figure 1.6 on p. 11):

$$\begin{aligned}\mathcal{R}_+[x, y, x', y'] \;:\Longleftrightarrow\; & [?\mathsf{0}(x) \wedge \mathit{false}] \vee \\ & [\neg\, ?\mathsf{0}(x) \wedge x' = {}^-(x) \wedge y' = y]\,.\end{aligned} \tag{5.16}$$

We have $(\mathsf{x}, \mathsf{y}) \succ_{\mathcal{R}_+,xy} (\mathsf{x}', \mathsf{y}')$ iff $\mathsf{x} = {}^+(\mathsf{x}')$ and $\mathsf{y} = \mathsf{y}'$. Clearly, the relation remains well-founded if we remove equation $y' = y$ in the range predicate of $\mathcal{R}_+$, because the x-component of $\succ_{\mathcal{R}^{opt}_+,xy}$ gets structurally smaller in each step:

$$\begin{aligned}\mathcal{R}^{opt}_+[x, x'] \;:\Longleftrightarrow\; & [?\mathsf{0}(x) \wedge \mathit{false}] \vee \\ & [\neg\, ?\mathsf{0}(x) \wedge x' = {}^-(x)]\,.\end{aligned} \tag{5.17}$$

The process of removing an equation from the range predicate is called *range generalization* in [85, 89].

One can also eliminate literals from the domain clause, which is called *domain generalization* [85, 89]. In the relation representation

$$\begin{aligned}\mathcal{R}_{-}[x,y,x',y'] \;:\Longleftrightarrow\; & [?0(x) \wedge \mathit{false}] \vee \\ & [\neg\, ?0(x) \wedge ?0(y) \wedge \mathit{false}] \vee \\ & [\neg\, ?0(x) \wedge \neg\, ?0(y) \wedge x' = {}^{-}(x) \wedge y' = {}^{-}(y)]\end{aligned} \tag{5.18}$$

of procedure "$-$" (cf. Figure 3.3 on p. 70), one can eliminate equation $y' = {}^{-}(y)$ by a range generalization and then eliminate literal $\neg\, ?0(y)$ by a domain generalization without affecting the well-foundedness of the relation representation:

$$\begin{aligned}\mathcal{R}'_{-}[x,y,x',y'] \;:\Longleftrightarrow\; & [?0(x) \wedge \mathit{false}] \vee \\ & [\neg\, ?0(x) \wedge ?0(y) \wedge \mathit{false}] \vee \\ & [\neg\, ?0(x) \wedge x' = {}^{-}(x)]\end{aligned}$$

The second atomic relation representation $\neg\, ?0(x) \wedge ?0(y) \wedge \mathit{false}$ is subsumed by the third one, because $\neg\, ?0(x) \wedge ?0(y) \rightarrow \neg\, ?0(x)$, and thus can be removed to get the optimal relation representation

$$\begin{aligned}\mathcal{R}_{-}^{opt,x}[x,x'] \;:\Longleftrightarrow\; & [?0(x) \wedge \mathit{false}] \vee \\ & [\neg\, ?0(x) \wedge x' = {}^{-}(x)]\,.\end{aligned} \tag{5.19}$$

Obviously, the challenge is to eliminate exactly those domain literals and those equations in a range predicate that are *not* required to ensure that the relation representation remains well-founded. As the relation representation of procedure "$-$" shows, there may be more than one possibility to generalize the relation representation: We could as well have eliminated $x' = {}^{-}(x)$ and $\neg\, ?0(x)$ from $\mathcal{R}_{-}$ instead of $y' = {}^{-}(y)$ and $\neg\, ?0(y)$:

$$\begin{aligned}\mathcal{R}_{-}^{opt,y}[y,y'] \;:\Longleftrightarrow\; & [?0(y) \wedge \mathit{false}] \vee \\ & [\neg\, ?0(y) \wedge y' = {}^{-}(y)]\,.\end{aligned} \tag{5.20}$$

We speak of an *optimized* relation representation if all possible generalization steps that we describe in the following have been performed.

Domain and range generalization. Using the results from termination analysis, relation representations are optimized heuristically. The heuristic eliminates literals from a relation representation that were not used in the termination proof. Clearly, such unused literals have no influence on the well-foundedness of the relation representation, so it is safe to eliminate them. This optimization is only a heuristic, because a suboptimal termination proof may have used more literals than necessary. However, this heuristic works well in practice.

Let $I_{used} \subseteq \{1, \ldots, n\}$ be the subset of the parameter indices that occur in measure term `m` if termination has been proved interactively (cf. Section 4.1) or that were considered in an automated termination proof (cf. Section 4.5), respectively. Furthermore, let $C_{used} \subseteq C_1 \cup \ldots \cup C_k$ be the subset of the literals that were used in the termination proof, where $C_1, \ldots, C_k$ are the call contexts that were considered.

Relation representation $\mathcal{R}_{proc}$ is optimized as follows [85, 89]:

- For each atomic relation representation $\bigwedge D \wedge R$ of $\mathcal{R}_{proc}$, remove all equations $x'_i = t_i$ from R with $i \notin I_{used}$.
- For each atomic relation representation $\bigwedge D \wedge R$ of $\mathcal{R}_{proc}$ with $R \neq \mathit{false}$, remove all domain literals d from D with $d \notin C_{used}$.
- Remove each atomic relation representation $\bigwedge D \wedge \mathit{false}$ from $\mathcal{R}_{proc}$ if there is an atomic relation representation $\bigwedge D' \wedge R'$ in $\mathcal{R}_{proc}$ with $D' \subseteq D$.

Example 5.23. In Example 4.65 (p. 127), the termination proof of procedure *map* considered only parameter k. Thus we eliminate $f' = \lambda y.\, f(y)$ from the relation representation of *map* (cf. Example 5.21) and get

$$\begin{aligned}\mathcal{R}^{opt}_{map}[k, k'] \;:\Longleftrightarrow\; & [?\varepsilon(k) \;\wedge\; \mathit{false}] \vee \\ & [\neg\, ?\varepsilon(k) \;\wedge\; k' = tl(k)]\,.\end{aligned}$$

This relation representation is optimal, because each removal of a literal from $\mathcal{R}^{opt}_{map}$ would destroy well-foundedness. ♢

Example 5.24. In Example 4.69 (p. 129) we proved termination of procedure *forall.list*. Before optimization, the relation representation of *forall.list* is

$$\begin{aligned}\mathcal{R}_{forall.list}[p, k, p', k'] \;:\Longleftrightarrow\; & \\ & [?\varepsilon(k) \;\wedge\; \mathit{false}] \vee \\ & [?::(k) \;\wedge\; p(hd(k)) \;\wedge\; p' = \lambda y.\, p(y) \;\wedge\; k' = tl(k)] \vee \\ & [?::(k) \;\wedge\; \neg\, p(hd(k)) \;\wedge\; \mathit{false}]\,.\end{aligned}$$

The termination proof only considered parameter k and used the literal(s) $C_{used} = \{?::(k)\}$. Thus we eliminate literal $p(hd(k))$ from the second atomic relation representation. Then the third atomic relation representation is removed, because it is subsumed by the second one. The resulting relation representation is

$$\begin{aligned}\mathcal{R}^{opt}_{forall.list}[k, k'] \;:\Longleftrightarrow\; & [?\varepsilon(k) \wedge \mathit{false}] \vee \\ & [?::(k) \wedge k' = tl(k)]\,,\end{aligned}$$

which again is optimal. ♢

In the following we describe our new optimization heuristics for procedures with second-order recursion.

Generalization of quantification procedures. The relation representation of procedure *groundterm* (cf. Example 5.20 on p. 144) uses quantification procedure *exists.every*, because *groundterm* is defined by second-order recursion using procedure *every*. In the termination proof of *groundterm* we used the fact that procedure *every* is call-bounded. The proof that *every* is call-bounded does not use condition $p(hd(k))$, cf. Example 4.53 (p. 120). This means that procedure *every* would remain call-bounded if the recursive call $every(p, tl(k))$ was also executed under condition $\neg\, p(hd(k))$. In other words, *groundterm* would terminate as well if we replaced the call of procedure *every* with a call of procedure *every′* shown in Figure 5.4. (The semantics of *groundterm* would change, of course, but we are only interested in termination here.)

The relation representation for this modified implementation of procedure *groundterm* would use quantification procedure *forall.every′*, see Figure 5.4.[5] This quantification procedure can be simplified by removing the irrelevant case analysis over $p(hd(k))$. Then parameter p is not used anymore, so we can remove it and get the optimized quantification procedure $forall^{opt}.every$ shown in Figure 5.4.

Procedure $forall^{opt}.every$ generalizes procedure *forall.every* as follows:

- It checks $p'(z)$ whenever *forall.every* checks $p'(z)$.
- It additionally checks $p'(z)$ for some more $z : @A$ that satisfy $\#(z, \epsilon) \leq \#(k, \mathbf{1})$. In fact, it checks $p'(z)$ for *all* items z of list k. Consequently, $forall^{opt}.every(p, k) \approx forall.list(p, k)$.

Since *every′* is call-bounded, it is sound to replace *exists.every* with a corresponding call of $exists^{opt}.every$ in the relation representation of *groundterm*; as described in Chapter 3, we write $exists^{opt}.every(p, k)$ as an abbreviation for $\neg\, forall^{opt}.every(\lambda z : @A.\, \neg\, p(z),\ k)$:

$$\begin{aligned}
&\mathcal{R}'_{groundterm}[t, t'] \ :\Longleftrightarrow \\
&\quad [?\, var(t) \wedge false] \vee \\
&\quad [?\, apply(t) \wedge exists^{opt}.every(\lambda s : term[@\,V, @F].\, t' = s,\ args(t))]\,.
\end{aligned}$$

This generalization of the range predicate is a significant benefit (see also Section 7.2.1), because $\mathcal{R}'_{groundterm}$ describes the *direct subterm* relation on $term[@\,V, @F]$ and thus is equivalent to the relation representations in Examples 5.13 (p. 140) and 5.19 (p. 144).

[5] Recall that *exists.every* just abbreviates a call of quantification procedure *forall.every*.

```
procedure every(p : @A → bool, k : list[@A]) : bool <=
if ?ε(k)
  then true
  else if p(hd(k))
         then every(p, tl(k))
         else false
       end
end

procedure every′(p : @A → bool, k : list[@A]) : bool <=
if ?ε(k)
  then true
  else if p(hd(k))
         then every′(p, tl(k))
         else every′(p, tl(k))
       end
end

procedure forall.every′(p′, p : @A → bool, k : list[@A]) : bool <=
if ?ε(k)
  then true
  else if p′(hd(k))
         then if p(hd(k))
                then forall.every′(p′, p, tl(k))
                else forall.every′(p′, p, tl(k))
              end
         else false
       end
end

procedure forall^opt.every(p′ : @A → bool, k : list[@A]) : bool <=
if ?ε(k)
  then true
  else if p′(hd(k))
         then forall^opt.every(p′, tl(k))
         else false
       end
end
```

Figure 5.4: Optimization of the quantification procedure for *every*

Definition 5.25 (Optimized quantification procedures)**.** *If the second-order procedure*

$$\begin{array}{l}\texttt{procedure}\ \mathit{proc}(f:\tau_1\times\ldots\times\tau_m\to\tau_f,\ x:\tau_x):\tau_{\mathit{proc}}\ <= \\ \texttt{assume}\ c_{\mathit{proc}};\ B_{\mathit{proc}}\end{array}$$

is (π, r, ϱ)*-call-bounded for some* $r \in \{1, \ldots, m\}$*, the* optimized quantification procedure $\mathit{forall}^{\mathit{opt}}_{\pi,r,\varrho}.\mathit{proc}$ for *proc is synthesized as follows:*

1. *Procedure*

$$\begin{array}{l}\texttt{procedure}\ \mathit{forall}^{\mathit{opt}}_{\pi,r,\varrho}.\mathit{proc}(p:\tau_r\to \mathit{bool}, \\ \qquad\qquad\qquad\qquad f:\tau_1\times\ldots\times\tau_m\to\tau_f, \\ \qquad\qquad\qquad\qquad x:\tau_x):\mathit{bool}\end{array}$$

   is derived from forall.proc by replacing all subterms $p(t_1, \ldots, t_m)$ *in the procedure body with* $p(t_r)$*.*

2. *For all conditions c that were* not *used in the proof that proc is call-bounded, the case analysis over c in the body of* $\mathit{forall}^{\mathit{opt}}_{\pi,r,\varrho}.\mathit{proc}$ *is replaced with the conjunction of its branches.*

3. *Each unused parameter of* $\mathit{forall}^{\mathit{opt}}_{\pi,r,\varrho}.\mathit{proc}$ *is removed.*

Example 5.26. Procedure *foldl* (cf. Figure 1.4 on p. 7) is $(\mathbf{1}, 2, \epsilon)$-call-bounded (cf. Example 4.54 on p. 121). We construct the optimized quantification procedure $\mathit{forall}^{\mathit{opt}}.\mathit{foldl}$ according to Definition 5.25:

1. We start with procedure

$$\begin{array}{l}\texttt{procedure}\ \mathit{forall.foldl}'(p:@B\to\mathit{bool},\ f:@A\times@B\to@A, \\ \qquad\qquad\qquad\qquad x:@A,\ k:\mathit{list}[@B]):\mathit{bool}\ <= \\ \mathit{if}\ ?\varepsilon(k) \\ \quad \mathit{then}\ \mathit{true} \\ \quad \mathit{else}\ \ \mathit{if}\ p(\mathit{hd}(k)) \\ \qquad\quad \mathit{then}\ \mathit{forall.foldl}'(p, f, f(x, \mathit{hd}(k)), \mathit{tl}(k)) \\ \qquad\quad \mathit{else}\ \ \mathit{false} \\ \qquad \mathit{end} \\ \mathit{end}\,.\end{array}$$

2. Condition $\neg\, ?\varepsilon(k)$ has been used in the proof that *foldl* is call-bounded, so the body of procedure $\mathit{forall}^{\mathit{opt}}.\mathit{foldl}$ remains unchanged in this step.

3. Parameter x is unused, because it only occurs in the x-argument $f(x, \mathit{hd}(k))$ of the recursive call. Thus it can be removed. Then parameter f is unused and can be removed as well. This yields the optimized quantification procedure

```
procedure forall^opt.foldl(p:@B → bool, k:list[@B]) : bool <=
if ?ε(k)
  then true
  else if p(hd(k))
         then forall^opt.foldl(p, tl(k))
         else false
       end
end .
```

Obviously, $forall^{opt}.foldl(p,k) \approx forall.list(p,k)$. ♢

Optimized quantification procedures in range predicates. If termination of a procedure f has been proved using call-boundedness of a second-order procedure *proc*, the relation representation of f remains well-founded if we replace *exists.proc* with $exists^{opt}.proc$. An optimized quantification procedure $forall^{opt}.proc$ is often[6] equivalent to a quantification procedure *forall.str* (in the sense that $forall^{opt}.proc(p,x) \approx forall.str(p,x)$), so we replace $exists^{opt}.proc$ with *exists.str* in the relation representation in these cases.

Equivalence of quantification procedures is determined by a simple heuristic: The formal parameters need to be a permutation of each other (up to renaming) and the bodies need to be syntactically equal up to a straightforward translation between *if*- and *case*-expressions.

Example 5.27. By optimizing the relation representations of procedures *varcount* and *groundterm* (cf. Examples 5.19 and 5.20 on p. 144) we get

$$\begin{aligned}\mathcal{R}^{opt}_{varcount}[t,t'] \iff \mathcal{R}^{opt}_{groundterm}[t,t'] \iff \\ [?var(t) \wedge false] \vee \\ [?apply(t) \wedge exists.list(\lambda s : term[@V, @F].\, t' = s,\ args(t))]\,.\end{aligned}$$

This relation representation is (up to a renaming of the variables) equal to the relation representation of *term*, cf. Example 5.13 (p. 140). ♢

Further optimization techniques. We refer to [73, 84, 85, 89] for further optimization techniques. These techniques ensure that for atomic relation representations $\bigwedge D_1 \wedge R_1$ and $\bigwedge D_2 \wedge R_2$ in a composed relation representation $\mathcal{R}$, either $D_1 = D_2$ or D_1 and D_2 exclude each other; i. e., if $D_1 \neq D_2$, then there are no values $q_1, \ldots, q_n$ with $eval_P(\bigwedge D_1) = true$ and $eval_P(\bigwedge D_2) = true$. This property is called *separation* and ensures that

[6] In general $forall^{opt}.proc$ is equivalent to *forall.str* if *proc* applies its first-order parameter to the items of another parameter. For instance, *map* applies the first-order parameter to all items of list k, so $forall^{opt}.map$ is equivalent to *forall.list*.

the atomic relation representations do not overlap, which would result in redundant proof obligations in inductive proofs.

Furthermore, negated structure predicates like $\neg\, ?\mathsf{0}(n)$ are converted to positive literals $?^{+}(n)$, which simplifies the resulting proof obligations.

5.3 Synthesis of Induction Formulas

If $\mathcal{R}$ is a well-founded relation representation over Σ and $\mathcal{V} \subseteq \{x_1, \ldots, x_n\}$, the principle of well-founded induction guarantees that

$$\forall \vec{x}.\ \big(\forall \vec{x'}.\ \mathcal{R}[\vec{x}, \vec{x'}] \to \mathit{goal}[\vec{x'}]\big) \to \mathit{goal}[\vec{x}] \tag{5.21}$$

implies $\forall \vec{x}.\ \mathit{goal}[\vec{x}]$ for any term $\mathit{goal} \in \mathcal{T}(\Sigma, \{x_1, \ldots, x_n\})$ with $\mathcal{V}_{\mathsf{f}}(\mathit{goal}) = \{x_1, \ldots, x_n\}$.

If $\mathcal{R} = A_1 \vee \ldots \vee A_k$ is a case complete relation representation with $A_i = \bigwedge D_i \wedge R_i$ for $i = 1, \ldots, k$, induction formula (5.21) can be split up into k induction formulas, one for each case $D_1, \ldots, D_k$:

$$\forall \vec{x}.\ D_i \wedge \big(\forall \vec{x'}.\ R_i[\vec{x}, \vec{x'}] \to \mathit{goal}[\vec{x'}]\big) \to \mathit{goal}[\vec{x}] \tag{5.22}$$

Example 5.28. We can prove $\forall n : \mathbb{N}.\ \mathit{goal}[n]$ by well-founded induction wrt. $\mathcal{R}_{\mathit{even}}$ (cf. p. 142) by showing each of the following induction formulas (we omit the leading quantification "$\forall n : \mathbb{N}$"):

- $?\mathsf{0}(n) \wedge (\forall n' : \mathbb{N}.\ \mathit{false} \to \mathit{goal}[n']) \to \mathit{goal}[n]$
- $\neg\, ?\mathsf{0}(n) \wedge ?\mathsf{0}(\mathsf{^-}(n)) \wedge (\forall n' : \mathbb{N}.\ \mathit{false} \to \mathit{goal}[n']) \to \mathit{goal}[n]$
- $\neg\, ?\mathsf{0}(n) \wedge \neg\, ?\mathsf{0}(\mathsf{^-}(n)) \wedge (\forall n' : \mathbb{N}.\ n' = \mathsf{^-}(\mathsf{^-}(n)) \to \mathit{goal}[n']) \to \mathit{goal}[n]$

The induction hypothesis of the base cases is "$\mathit{false} \to \mathit{goal}[n']$" and thus is equivalent to *true*. For the step case, the induction hypothesis

$$\forall n' : \mathbb{N}.\ n' = \mathsf{^-}(\mathsf{^-}(n)) \to \mathit{goal}[n']$$

is equivalent to $\mathit{goal}[\mathsf{^-}(\mathsf{^-}(n))]$. Hence we reformulate the induction formulas:

- $?\mathsf{0}(n) \to \mathit{goal}[n]$
- $\neg\, ?\mathsf{0}(n) \wedge ?\mathsf{0}(\mathsf{^-}(n)) \to \mathit{goal}[n]$
- $\neg\, ?\mathsf{0}(n) \wedge \neg\, ?\mathsf{0}(\mathsf{^-}(n)) \wedge \mathit{goal}[\mathsf{^-}(\mathsf{^-}(n))] \to \mathit{goal}[n]$. ♢

Example 5.29. We can prove $\forall t : \mathit{term}[@\,V, @F].\ \mathit{goal}[t]$ by well-founded induction wrt. $\mathcal{R}_{\mathit{term}}$ (cf. Example 5.13 on p. 140) by showing each of the following induction formulas (where we again omit the leading quantification "$\forall t : \mathit{term}[@\,V, @F]$"):

- $?var(t) \wedge (\forall t' : term[@\,V, @F].\ false \rightarrow goal[t']) \rightarrow goal[t]$
- $?apply(t) \wedge$
  $\big(\forall t' : term[@\,V, @F].$
  $exists.list(\lambda s : term[@\,V, @F].\, t' = s,\ args(t)) \rightarrow goal[t']\big) \rightarrow goal[t]$

We can simplify away the induction hypothesis of the base case as in the previous example. For the step case the induction hypothesis states that $goal[t']$ is true whenever t' is an item of list $args(t)$. Consequently, $goal[s]$ is true for all items s of list $args(t)$ and we can reformulate the induction formulas into the following more readable form:

- $?var(t) \rightarrow goal[t]$
- $?apply(t) \wedge forall.list(\lambda s : term[@\,V, @F].\, goal[s],\ args(t)) \rightarrow goal[t]$

Note that this is the usual induction axiom for terms: We need to show $goal[t]$ for the case that t is a variable and we need to show $goal[t]$ for the case that t is of the form $apply(f, t_1 :: \ldots :: t_n :: \varepsilon)$, where we may assume that $goal[t_i]$ holds for all $i = 1, \ldots, n$. ♢

We generally instantiate induction hypotheses as in the examples: For a range predicate R over Σ and $\mathcal{V} = \{x_1, \ldots, x_n\}$ and a goal term $goal \in \mathcal{T}(\Sigma(P), \{x_1, \ldots, x_n\})$, we define the instantiation of the primed goal term $goal' := goal[x'_1, \ldots, x'_n]$ wrt. R by

$$
\begin{aligned}
&Inst(false,\ goal') := \emptyset \\
&Inst\big(\textstyle\bigwedge_{i=1}^{n} x'_i = t_i,\ goal'\big) := \{goal'[x'_1/t_1, \ldots, x'_n/t_n]\} \\
&Inst\big(exists(\lambda y_1 : \tau_1, \ldots, y_m : \tau_m.\, \textstyle\bigwedge D' \wedge R',\ t_1, \ldots, t_k),\ goal'\big) := \\
&\qquad \big\{forall(\lambda y_1 : \tau_1, \ldots, y_m : \tau_m.\, \textstyle\bigwedge D' \wedge Inst(R', goal'),\ t_1, \ldots, t_k)\big\}\,.
\end{aligned}
$$

As outlined in Section 5.1, each induction formula (5.22) is represented by a sequent:

$$\langle D_i, Inst(R_i, goal[x'_1, \ldots, x'_n]) \Vdash goal[x_1, \ldots, x_n]\rangle$$

In general, some D_i may be equal. The following definition then takes the union of all $Inst(R_i, goal[x'_1, \ldots, x'_n])$ with $D_i = D$ as induction hypotheses for some $D \in \{D_1, \ldots, D_k\}$.

Definition 5.30 (Induction formulas)**.** *Let $goal \in \mathcal{T}(\Sigma(P), \{x_1, \ldots, x_n\})_{bool}$ for some $\mathcal{L}$-program P such that $\mathcal{V}_{\mathsf{f}}(goal) = \{x_1, \ldots, x_n\}$. Furthermore, let $\mathcal{R} = A_1 \vee \ldots \vee A_k$ be a well-founded relation representation over $\Sigma(P)$ and $\mathcal{V} \subseteq \{x_1, \ldots, x_n\}$ with $A_i = \bigwedge D_i \wedge R_i$ for $i = 1, \ldots, k$.*

For each $H \in \mathcal{CL}(\Sigma, \mathcal{V})$ we define the induction formula *for goal wrt. H and $\mathcal{R}$ by $IndForm_{\mathcal{R}}(H, goal) := \langle H, IH \Vdash goal\rangle$, where*

$$IH := \bigcup_{\substack{i \in \{1,\ldots,k\} \\ \text{and } D_i = H}} Inst(R_i, goal[x'_1, \ldots, x'_n])\,.$$

The set $IndForm_{\mathcal{R}}(goal)$ *of* induction formulas for *goal* wrt. $\mathcal{R}$ *is defined by*

$$IndForm_{\mathcal{R}}(goal) := \{\, IndForm_{\mathcal{R}}(D, goal) \mid D \in \{D_1, \ldots, D_k\}\} \ .$$

For a sequent $\langle \emptyset, \emptyset \Vdash goal\rangle$ and a relation representation $\mathcal{R}$, the proof rule *Induction* of the HPL-calculus (cf. p. 134) generates a child node for each induction formula in $IndForm_{\mathcal{R}}(goal)$.[7] Each child sequent $\langle H, \emptyset \Vdash goal\rangle$ represents a base case of the induction and each sequent $\langle H, IH \Vdash goal\rangle$ with $IH \neq \emptyset$ in general represents a step case of the induction.

In Example 5.29, $?var(t)$ clearly is a base case. Case $?apply(t)$ in general is a step case, because $t = apply(f, t_1 :: \ldots :: t_n :: \varepsilon)$ and the induction hypothesis ensures that $goal[t_i]$ for $i = 1, \ldots, n$. But n may be 0 (i. e., $args(t)$ may be empty). Strictly speaking, $n = 0$ is a base case, so the second induction formula covers both a base case, viz. $?apply(t) \wedge ?\varepsilon(args(t))$, and a step case, viz. $?apply(t) \wedge \neg\, ?\varepsilon(args(t))$. Typically the distinction between these two subcases of $?apply(t)$ is unimportant in a proof, so Definition 5.30 generates a single sequent for this case. Of course, the user is free to split this sequent up into the subcases $?\varepsilon(args(t))$ and $\neg\, ?\varepsilon(args(t))$ if this seems appropriate in a particular proof.

We conclude with an example that demonstrates how the induction axiom for *complete induction* on natural numbers (also called *strong induction* or *course of values induction*) is inferred from a procedure definition.

Example 5.31 (Complete induction)**.** Procedure *sum* in Figure 5.5 computes $f(0) + f(1) + \ldots + f(n)$. Procedure *zero* is defined by second-order recursion via *sum*. It returns $\mathtt{0}$ for all n, because $zero(n) = zero(0) + zero(1) + \ldots + zero(^-(n))$ and $zero(\mathtt{0}) = \mathtt{0}$. The optimized relation representation of procedure *zero* is given by:

$$\begin{aligned} \mathcal{R}^{opt}_{zero}[n, n'] \ :\Longleftrightarrow\ & [?\mathtt{0}(n) \wedge false] \vee \\ & [\neg\, ?\mathtt{0}(n) \wedge exists^{opt}.sum(\lambda m : \mathbb{N}.\, n' = m,\ ^-(n))] \ . \end{aligned}$$

Since $exists^{opt}.sum(\lambda m : \mathbb{N}.\, n' = m,\ ^-(n)) \approx (n > n')$, the induction formulas

- $\langle \{?\mathtt{0}(n)\}, \emptyset \Vdash goal[n]\rangle$ and
- $\langle \{\neg\, ?\mathtt{0}(n)\}, \{forall^{opt}.sum(\lambda m : \mathbb{N}.\, goal[m],\ ^-(n))\} \Vdash goal[n]\rangle$

represent the induction axiom that one can prove $\forall n : \mathbb{N}.\ goal[n]$ by showing

- $goal[\mathtt{0}]$ and
- $goal[n]$ for $n \neq \mathtt{0}$ with the induction hypothesis that $goal[m]$ for all $m < n$. ♢

[7] The induction hypotheses *IH* are simplified by symbolic evaluation before the corresponding sequent is created [73]. See Section 5.5 for an example. Proof rule *Induction* is sound, because Definition 5.30 assumes a *well-founded* relation representation $\mathcal{R}$.

```
procedure sum(f : ℕ → ℕ, n : ℕ) : ℕ <=
if ?0(n)
  then f(0)
  else f(n) + sum(f, ⁻(n))
end

procedure forall.sum(p : ℕ → bool, f : ℕ → ℕ, n : ℕ) : bool <=
if ?0(n)
  then p(0)
  else if p(n)
         then forall.sum(p, f, ⁻(n))
         else false
       end
end

procedure forall^opt.sum(p : ℕ → bool, n : ℕ) : bool <=
if ?0(n)
  then p(0)
  else if p(n)
         then forall^opt.sum(p, ⁻(n))
         else false
       end
end

procedure zero(n : ℕ) : ℕ <=
if ?0(n)
  then 0
  else sum(zero, ⁻(n))
end
```

Figure 5.5: An example of second-order recursion which leads to an induction axiom for complete induction

5.4 Symbolic Evaluation

Symbolic evaluation generalizes the evaluation by the interpreter $eval_P$ (cf. Section 2.3.1) by allowing the term to contain variables.[8] For instance, a term $?cons_i(cons_i(t_1, \ldots, t_{n_i}))$ is symbolically evaluated to *true* even if $t_1, \ldots, t_n \notin \mathbb{V}(P)$. Symbolic evaluation is defined by the *evaluation calculus*, which comprises more than 50 inference rules (called *evaluation rules*) and is discussed in detail in [73, 89, 98]. The presentation by Schweitzer [73] annotates the evaluation rules with additional heuristics to guide a prover (based on this calculus) to "useful" simplifications; for the evaluation rules that we added to handle second-order programs, no such heuristics are necessary.

The evaluation calculus is parameterized with a set $H \in \mathcal{CL}(\Sigma(P), \mathcal{V})$ of *hypotheses* and a finite set $\mathcal{A} \subset \mathcal{CL}(\Sigma(P), \mathcal{V} \cup \mathcal{V}')$ of *assumptions*. We write $t \Rightarrow_{P,H,\mathcal{A}} t'$ iff t' results from t by applying some evaluation rule. Each hypothesis $h \in H$ and each (disjunctive) clause $A \in \mathcal{A}$ may be assumed to be true when symbolically evaluating term t. The hypotheses H denote the context of the symbolic evaluation and comprise the *global hypotheses* H_g of the corresponding sequent $\langle H_g, IH \Vdash goal \rangle$ as well as the *local hypotheses* H_l, given by the conditions that lead to subterm t of *goal*. The assumptions $\mathcal{A}$ are given by the clauses of those lemmas that possess status verified and the clauses of the induction hypotheses IH. Primed variables denote universally quantified variables and may be instantiated with arbitrary terms.

Figure 5.6 shows some evaluation rules to give an impression how these rules look like (we omit indices H and $\mathcal{A}$ in side conditions $t \Rightarrow_{P,H,\mathcal{A}} t'$ if these sets are left unchanged). There are four categories of evaluation rules:

Generalized computation rules: These evaluation rules generalize inference rules of the computation calculus (cf. Figure 2.6 on p. 51) by relaxing the side conditions. For instance, the evaluation rules *Affirmative structure test* and *Appropriate selector* generalize computation rules (1) and (3).

There is also an evaluation rule that generalizes computation rule (15):

(Execute procedure call)

$$\frac{proc(t_1, \ldots, t_n)}{if\{c_{proc}, B_{proc}, proc(x_1, \ldots, x_n)\}[x_1/t_1, \ldots, x_n/t_n]} \quad ,$$

if `EXECUTE?`$[proc(t_1, \ldots, t_n)]$

It replaces a procedure call with the instantiated procedure body, guarded by the context requirement, if this seems heuristically useful. The heuristic check is denoted by `EXECUTE?`[...] in the side condition.

[8] The variables are regarded as *skolemized variables*, i. e., constants.

(Affirmative structure test) $\dfrac{?cons_i(cons_i(t_1,\ldots,t_{n_i}))}{true}$

(Negative structure test) $\dfrac{?cons_{i'}(cons_i(t_1,\ldots,t_{n_i}))}{false}$, if $cons_{i'} \neq cons_i$

(Appropriate selector) $\dfrac{sel_{i,j}(cons_i(q_1,\ldots,q_{n_i}))}{q_j}$

(Reflexivity) $\dfrac{t_1 = t_2}{true}$, if $t_1 \simeq t_2$

(Constructor uniqueness) $\dfrac{cons_i(\ldots) = cons_{i'}(\ldots)}{false}$, if $i' \neq i$

(Constructor injectivity) $\dfrac{cons_i(t_1,\ldots,t_{n_i}) = cons_i(t'_1,\ldots,t'_{n_i})}{t_1 = t'_1 \wedge \ldots \wedge t_{n_i} = t'_{n_i}}$

(Keep then-part) $\dfrac{if\{true,t_1,t_2\}}{t_1}$

(Keep else-part) $\dfrac{if\{false,t_1,t_2\}}{t_2}$

(Skip alternatives) $\dfrac{if\{b,true,false\}}{b}$

(Skip condition) $\dfrac{if\{b,t,t\}}{t}$

(Evaluate condition) $\dfrac{if\{b,t_1,t_2\}}{if\{b',t_1,t_2\}}$, if $b \Rightarrow_P b'$

(Evaluate then-part) $\dfrac{if\{b,t_1,t_2\}}{if\{b,t'_1,t_2\}}$, if $t_1 \Rightarrow_{P,H\cup\{b\}} t'_1$

(Evaluate else-part) $\dfrac{if\{b,t_1,t_2\}}{if\{b,t_1,t'_2\}}$, if $t_2 \Rightarrow_{P,H\cup\{\neg b\}} t'_2$

(Evaluate argument) $\dfrac{f(t_1,\ldots,t_{i-1},t_i,t_{i+1},\ldots,t_n)}{f(t_1,\ldots,t_{i-1},t'_i,t_{i+1},\ldots,t_n)}$, if $t_i \Rightarrow_P t'_i$, $f \notin \Sigma(P)^{\mathsf{cond}}$

(Evaluate let-body) $\dfrac{let\{x := t_1;\ t_0\}}{let\{x := t_1;\ t'_0\}}$, if $t_0 \Rightarrow_P t'_0$

(Affirmative hypothesis) $\dfrac{a}{true}$, if $a \in \mathcal{AT}(\Sigma(P),\mathcal{V})$ and $a \in_{\simeq} H$

(Negative hypothesis) $\dfrac{a}{false}$, if $a \in \mathcal{AT}(\Sigma(P),\mathcal{V})$ and $\neg\, a \in_{\simeq} H$

Figure 5.6: Some inference rules of the evaluation calculus

This evaluation rule is very complex, because it needs to trade off the following objectives:

- Procedure calls shall be unfolded sufficiently often in order to be able to exploit their definition.
- Symbolic evaluation is supposed to terminate so that the theorem prover comes up with a result within reasonable time. Thus procedure calls may only be unfolded finitely often.
- The result of symbolic evaluation should be "useful" so that the user can continue the proof based on the result. Unfolding a procedure too often blows up the goal term and prevents the application of lemmas and induction hypotheses.

Derived rules: Evaluation rules such as *Reflexivity*, *Skip alternatives*, and *Skip condition* are derived from properties of the interpreter $eval_P$. For instance, *Reflexivity* evaluates an equation $t_1 = t_2$ to *true* if t_1 and t_2 are equal up to a permutation of the arguments of commutative and/or associative function symbols [78] and α-conversion (i. e., a renaming of the variables introduced by λ-expressions) [99]; we denote this notion of equality by $t_1 \simeq t_2$.

Evaluate then-part and *Evaluate else-part* facilitate the evaluation of the branches t_1 and t_2 of an *if*-expression even if condition b cannot be evaluated to *true* or *false*. The extension of the set H of hypotheses takes truth or falsity of b into account when evaluating the alternatives. Thus an occurrence of b in t_1 or t_2 can be evaluated to *true* or *false* using evaluation rules *Affirmative hypothesis* or *Negative hypothesis*, respectively.

Assumption rules: For each assumption $A = \{lit, lit'_1, \ldots, lit'_n\} \in \mathcal{A}$, symbolic evaluation may assume *lit* as true if all literals $lit'_1, \ldots, lit'_n$ can be shown to be false, i. e., if $\mathsf{NOR}(A \setminus \{lit\})$ is true. Evaluation rule *Affirmative assumption* (cf. Figure 5.7) implements this reasoning step for an atom a that matches literal *lit* of an assumption A via some term substitution σ. This term substitution is determined automatically by a matching algorithm [78, 89, 99] and instantiates all universally quantified variables (i. e., all primed variables) in A.

For instance, if lemma

$$\texttt{lemma}\ \text{dbl(n) is even} <= \forall n : \mathbb{N}.\ even(dbl(n))$$

has been verified, then $A := \{even(dbl(n'))\} \in \mathcal{A}$ so a term of the form $even(dbl(t))$ is symbolically evaluated to *true*.

(Affirmative assumption) $\dfrac{a}{\mathsf{NOR}(\sigma(A) \setminus \{\sigma(lit)\})}$

if some $A \in \mathcal{A}$, some $lit \in A$, and some term substitution σ exist such that

(i) $a \in \mathcal{AT}(\Sigma(P), \mathcal{V})$,
(ii) $a \simeq \sigma(lit)$,
(iii) $dom(\sigma) \subseteq \mathcal{V}'$ and $\mathcal{V}_{\mathsf{f}}(\sigma(A)) \cap \mathcal{V}' = \emptyset$, and
(iv) $\sigma(lit') \Rightarrow_P^* false$ for all $lit' \in A \setminus \{lit\}$

(Negative assumption) $\dfrac{a}{\mathsf{OR}(\sigma(A) \setminus \{\sigma(lit)\})}$

if some $A \in \mathcal{A}$, some $lit \in A$, and some term substitution σ exist such that

(i) $a \in \mathcal{AT}(\Sigma(P), \mathcal{V})$,
(ii) $a \simeq \sigma(\overline{lit})$,
(iii) $dom(\sigma) \subseteq \mathcal{V}'$ and $\mathcal{V}_{\mathsf{f}}(\sigma(A)) \cap \mathcal{V}' = \emptyset$, and
(iv) $\sigma(lit') \Rightarrow_P^* false$ for all $lit' \in A \setminus \{lit\}$

Figure 5.7: Evaluation rules *Affirmative assumption* and *Negative assumption*

Negative assumption considers the dual case where atom a matches the complement $\overline{lit}$ of a literal lit.[9]

The assumption rules do not directly replace atom a with *true* or *false*, respectively, so that the subevaluations $\sigma(lit') \Rightarrow_P^* false$ are clearly visible in a deduction within the evaluation calculus; i. e., they are not hidden in the side condition of the rules. We use these evaluation rules in a more complex symbolic evaluation in Section 5.5.

Replacement rules: These evaluation rules reason about oriented equalities $t_1 \Rightarrow t_2$. For instance, if $\mathcal{A}$ contains an assumption

$$A = \{half(dbl(n')) \Rightarrow n'\},$$

then $half(dbl(t))$ is symbolically evaluated to t, because the equation $half(dbl(n')) \Rightarrow n'$ is oriented from left to right, see [98].

Formally, the evaluation calculus is defined as follows:

[9] The *complement* $\overline{lit}$ of a literal $lit \notin \{true, false\}$ is defined as $\neg\, lit$ if lit is a positive literal and as lit' if $lit = \neg\, lit'$.

Definition 5.32 (Evaluation calculus)**.** *For an $\mathcal{L}$-program P, a set $H \in \mathcal{CL}(\Sigma(P), \mathcal{V})$ of* hypotheses*, and a finite set $\mathcal{A} \subset \mathcal{CL}(\Sigma(P), \mathcal{V} \cup \mathcal{V}')$ of* assumptions*, the* evaluation calculus *is defined by:*

Language: $\mathcal{T}(\Sigma(P), \mathcal{V})$

Inference Rules: *The inference rules (called* evaluation rules*) are of the form*

$$\frac{t}{t'} \text{ , if } \mathsf{cond}[t, t']$$

such that $t' \in \mathcal{T}(\Sigma(P), \mathcal{V})_\tau$ whenever $t \in \mathcal{T}(\Sigma(P), \mathcal{V})_\tau$ for some type τ. An evaluation rule may only be applied if side condition $\mathsf{cond}[t, t']$ is satisfied.

Deduction: *We write $t \Rightarrow_{P,H,\mathcal{A}} t'$ iff t' results from t by applying some evaluation rule. $\Rightarrow^*_{P,H,\mathcal{A}}$ denotes the reflexive and transitive closure of $\Rightarrow_{P,H,\mathcal{A}}$. We omit the indices H and $\mathcal{A}$ if they are obvious from the context.*

Symbolic evaluation is invoked by proof rule *Simplification* of the HPL-calculus. It is used to prove the base and step cases in an inductive proof. In the following subsections we describe the evaluation rules that we added to reason about λ-expressions and second-order procedures.

5.4.1 β-Reduction

Figure 5.8 shows the straightforward generalization of computation rule (16) for β-reduction.

Example 5.33. $(\lambda x : @A. \neg p(x))(y) \Rightarrow_P \neg p(y)$ ♢

5.4.2 Evaluation of λ-Bodies

Evaluation rule *Evaluate λ-body* (Figure 5.8) is defined analogously to the existing evaluation rule *Evaluate let-body* (cf. Figure 5.6 and [73]).

Example 5.34.

$$\lambda n : \mathbb{N}. 1 + n \Rightarrow^*_P \lambda n : \mathbb{N}. {}^+(n) \,,$$

because

$$1 + n \Rightarrow^*_P {}^+(n) \ .$$ ♢

A further example is given in Section 5.4.4.

$$\text{(β-reduction)} \quad \frac{(\lambda x_1, \ldots, x_n.\, t)(t_1, \ldots, t_n)}{t[x_1/t_1, \ldots, x_n/t_n]}$$

$$\text{(Evaluate λ-body)} \quad \frac{\lambda x_1, \ldots, x_n.\, t}{\lambda x_1, \ldots, x_n.\, t'} \quad \text{, if } t \Rightarrow_P t'$$

$$\text{(Trivial \textit{forall}-call)} \quad \frac{\mathit{forall}(\lambda x_1, \ldots, x_n.\, \mathit{true},\ t_1, \ldots, t_m)}{\mathit{true}} \quad ,$$

$$\text{if } \mathit{forall} \in \Sigma(P)^{\mathsf{all}}$$

$$\text{(Distribute atom)} \quad \frac{h(\ldots,\ \lambda x_1, \ldots, x_n.\, t,\ \ldots)}{\begin{array}{l}\mathit{if}\{\mathit{CR}(t|_\pi),\\ \quad \mathit{if}\{t|_\pi,\\ \quad\quad h(\ldots,\ \lambda x_1, \ldots, x_n.\, t[\pi \leftarrow \mathit{true}],\ \ldots),\\ \quad\quad h(\ldots,\ \lambda x_1, \ldots, x_n.\, t[\pi \leftarrow \mathit{false}],\ \ldots)\},\\ \quad h(\ldots,\ \lambda x_1, \ldots, x_n.\, t,\ \ldots)\}\end{array}} \quad ,$$

$$\text{if } t|_\pi \in \mathcal{T}(\Sigma(P), \mathcal{V} \setminus \{x_1, \ldots, x_n\})_{bool} \setminus \{\mathit{true}, \mathit{false}\},$$
$$\Sigma_{\mathsf{f}}(t|_\pi) \cap \Sigma(P)^{\mathsf{cond}} = \emptyset, \text{ and } \mathit{CR}(t|_\pi) \Rightarrow_P^* \mathit{true}$$

(Distribute constant case condition)

$$\frac{h(\ldots,\ \lambda x_1, \ldots, x_n.\, t,\ \ldots)}{\begin{array}{l}\mathit{if}\{\mathit{CR}(t'),\\ \quad \mathit{case}\{t';\ \ldots, \mathit{cons}_i : h(\ldots,\ \lambda x_1, \ldots, x_n.\, t[\pi \leftarrow t_i],\ \ldots), \ldots\},\\ \quad h(\ldots,\ \lambda x_1, \ldots, x_n.\, t,\ \ldots)\}\end{array}} \quad ,$$

$$\text{if } t|_\pi = \mathit{case}\{t'; \mathit{cons}_1 : t_1, \ldots, \mathit{cons}_m : t_m\},$$
$$\mathcal{V}_{\mathsf{f}}(t') \cap \{x_1, \ldots, x_n\} = \emptyset, \text{ and } \mathit{CR}(t') \Rightarrow_P^* \mathit{true}$$

Figure 5.8: New evaluation rules for second-order features

5.4.3 Trivial Calls of Quantification Procedures

Since a quantification procedure *forall* returns *true* iff a predicate p is satisfied for all elements of a finite set, *forall* returns *true* if the predicate is equal to $\lambda x_1, \ldots, x_n.\, true$. The corresponding evaluation rule is shown in Figure 5.8. For example,

$$forall.list(\lambda x : @A.\, true,\ k) \ \Rightarrow_P \ \ true$$

and

$$forall.foldl(\lambda a : @A, b : @B.\, true,\ f,\ x,\ k) \ \Rightarrow_P \ \ true\,.$$

See Section 5.4.4 for a further example.

A corresponding evaluation rule for predicates $\lambda x_1, \ldots, x_n.\, false$ is not required, because in this case it suffices to unfold the procedure call:

$$\begin{aligned}
&forall.list(\lambda x : @A.\, false,\ k)\\
\Rightarrow^*_P\ &case\{k;\ \varepsilon : true,\ :: : if\{(\lambda x : @A.\, false)(hd(k)), \ldots, false\}\}\\
\Rightarrow^*_P\ &case\{k;\ \varepsilon : true,\ :: : false\}\\
\Rightarrow^*_P\ &?\varepsilon(k)
\end{aligned}$$

5.4.4 Extraction of Constants

Consider the procedure call

$$map(\lambda x.\, if\{b, f(x), g(x)\},\ k)\,.$$

If term variable x does not occur in b, then *map* will either apply f to all $x \in k$ or it will apply g to all $x \in k$, depending on b. Hence it is redundant to check b in each application of the λ-expression to some $x \in k$. We can instead check b before calling *map* and use the appropriate first-order parameter in each case:

$$if\{b, map(f, k), map(g, k)\}$$

This is the point of evaluation rule *Distribute atom* in Figure 5.8. Whenever the body of a λ-expression contains an atom $t|_\pi$ at some position π that does not use the variables $x_1, \ldots, x_n$ introduced by the λ-expression, then this is a "constant" atom, because it will evaluate to the same truth value in each call of the λ-expression. If the context requirement of this constant atom is also satisfied outside the λ-expression (which is ensured by side condition "$CR(t|_\pi) \Rightarrow^*_P true$"), then we can pull the case analysis on $t|_\pi$ out of the λ-expression.

Example 5.35.

$$forall.list(\lambda s : term[@V, @F].\ if\{?var(v), t_1, t_2\},\ args(t))$$

$\Rightarrow_P$ *(Distribute atom)*

$$\begin{array}{l} if\{true, \\ \quad if\{?var(v), \\ \quad\quad forall.list(\lambda s : term[@V, @F].\ if\{true, t_1, t_2\},\ args(t)), \\ \quad\quad forall.list(\lambda s : term[@V, @F].\ if\{false, t_1, t_2\},\ args(t))\}, \\ \quad forall.list(\lambda s : term[@V, @F].\ if\{?var(v), t_1, t_2\},\ args(t))\} \end{array}$$

$\Rightarrow_P$ *(Keep then-part)*

$$\begin{array}{l} if\{?var(v), \\ \quad forall.list(\lambda s : term[@V, @F].\ if\{true, t_1, t_2\},\ args(t)), \\ \quad forall.list(\lambda s : term[@V, @F].\ if\{false, t_1, t_2\},\ args(t))\} \end{array}$$

$\Rightarrow_P^*$ *(Evaluate then-part)+(Evaluate else-part)*

$$\begin{array}{l} if\{?var(v), \\ \quad forall.list(\lambda s : term[@V, @F].\ t_1,\ args(t)), \\ \quad forall.list(\lambda s : term[@V, @F].\ t_2,\ args(t))\} \end{array}$$

$\diamondsuit$

The following example illustrates the use of *Distribute atom* when the constant atom does not occur as condition of an *if*-expression:

Example 5.36.

$$forall.list(\lambda s : term[@V, @F].\ if\{p(s), true, ?var(v)\},\ args(t))$$

$\Rightarrow_P$ *(Distribute atom)*

$$\begin{array}{l} if\{true, \\ \quad if\{?var(v), \\ \quad\quad forall.list(\lambda s : term[@V, @F].\ if\{p(s), true, true\},\ args(t)), \\ \quad\quad forall.list(\lambda s : term[@V, @F].\ if\{p(s), true, false\},\ args(t))\}, \\ \quad forall.list(\lambda s : term[@V, @F].\ if\{p(s), true, ?var(v)\},\ args(t))\} \end{array}$$

$\Rightarrow_P$ *(Keep then-part)*

$$\begin{array}{l} if\{?var(v), \\ \quad forall.list(\lambda s : term[@V, @F].\ if\{p(s), true, true\},\ args(t)), \\ \quad forall.list(\lambda s : term[@V, @F].\ if\{p(s), true, false\},\ args(t))\} \end{array}$$

$\Rightarrow_P$ *(Evaluate then-part), (Evaluate λ-body), (Skip condition)*

$$\begin{aligned}&\mathit{if}\{?\mathit{var}(v),\\ &\quad \mathit{forall.list}(\lambda s : \mathit{term}[@\,V, @F].\, \mathit{true},\ \mathit{args}(t)),\\ &\quad \mathit{forall.list}(\lambda s : \mathit{term}[@\,V, @F].\, \mathit{if}\{p(s), \mathit{true}, \mathit{false}\},\ \mathit{args}(t))\}\end{aligned}$$

$\Rightarrow_P$ *(Evaluate then-part), (Trivial forall-call)*

$$\begin{aligned}&\mathit{if}\{?\mathit{var}(v),\\ &\quad \mathit{true},\\ &\quad \mathit{forall.list}(\lambda s : \mathit{term}[@\,V, @F].\, \mathit{if}\{p(s), \mathit{true}, \mathit{false}\},\ \mathit{args}(t))\}\end{aligned}$$

$\Rightarrow_P$ *(Evaluate else-part), (Evaluate λ-body), (Skip alternatives)*

$$\mathit{if}\{?\mathit{var}(v),\ \mathit{true},\ \mathit{forall.list}(\lambda s : \mathit{term}[@\,V, @F].\, p(s),\ \mathit{args}(t))\} \qquad \diamondsuit$$

Of course, a λ-expression can also contain a *case*-expression with a condition that is constant for each application of the λ-expression. Consider for instance the procedure call

$$\mathit{map}(\lambda x.\, \mathit{case}\,\{t'; \mathit{cons}_1 : f_1(x), \ldots, \mathit{cons}_n : f_n(x)\},\ k)\,.$$

If $x \notin \mathcal{V}_{\mathsf{f}}(t')$, then we can pull out the case analysis and get

$$\mathit{case}\,\{t'; \mathit{cons}_1 : \mathit{map}(f_1, k), \ldots, \mathit{cons}_n : \mathit{map}(f_n, k)\}\,.$$

The corresponding evaluation rule *Distribute constant case condition* is displayed in Figure 5.8.

5.5 An Example Proof

In this section we demonstrate the interplay of the techniques from the previous sections with a small example. Consider the following statement:

> "A ground term does not contain a variable as subterm."

To formalize this statement, we use procedures *groundterm* and *subterm* of Figure 1.5 (p. 9) and state the $\mathcal{L}$-lemma

$$\begin{aligned}&\texttt{lemma}\ \text{variable not subterm of groundterm} <= \forall v, t : \mathit{term}[@\,V, @F].\\ &\mathit{if}\{?\mathit{var}(v), \mathit{if}\{\mathit{groundterm}(t), \neg\, \mathit{subterm}(v, t), \mathit{true}\}, \mathit{true}\}\end{aligned}$$

The initial sequent of the proof tree of this lemma is $\langle \emptyset, \emptyset \Vdash \mathit{goal}[v, t]\rangle$ for the goal term

$$\mathit{goal}[v, t] := \mathit{if}\{?\mathit{var}(v), \mathit{if}\{\mathit{groundterm}(t), \neg\, \mathit{subterm}(v, t), \mathit{true}\}, \mathit{true}\}\,.$$

In the following subsections we describe how our approach constructs a proof tree for this lemma. For the step case, one auxiliary lemma is required:

$$
\begin{array}{l}
\texttt{lemma}\ \text{“}[\forall x \in k.\ p(x) \Rightarrow \neg\, q(x)] \wedge [\forall x \in k.\ p(x)] \Rightarrow \\
\qquad\qquad [\neg\, \exists x \in k.\ q(x)]\text{”}\ \ <= \\
\forall p, q : @A \rightarrow \mathit{bool}, k : \mathit{list}[@A]. \\
\ \mathit{if}\{\mathit{forall.list}(\lambda x : @A.\ \mathit{if}\{p(x), \neg\, q(x), \mathit{true}\},\ k), \\
\quad \mathit{if}\{\mathit{every}(p, k), \neg\, \mathit{some}(q, k), \mathit{true}\}, \\
\quad \mathit{true}\}
\end{array}
\tag{5.23}
$$

Apart from that, no interaction is needed; i. e., if the auxiliary lemma is verified before starting the proof of lemma “variable not subterm of groundterm”, then *no* user interaction is needed. We discuss a simplified version of the auxiliary lemma in Section 7.2.

5.5.1 Synthesis of Induction Formulas

We use the optimized relation representations of *groundterm* and *subterm* (cf. Example 5.27 on p. 152):

$$
\begin{array}{l}
\mathcal{R}^{opt}_{groundterm}[t, t'] \iff \mathcal{R}^{opt}_{subterm}[t, t'] \iff \\
\qquad [?\mathit{var}(t) \wedge \mathit{false}] \vee \\
\qquad [?\mathit{apply}(t) \wedge \mathit{exists.list}(\lambda s : \mathit{term}[@\,V, @F].\ t' = s,\ \mathit{args}(t))]\,.
\end{array}
$$

Proof rule *Induction* generates the following induction formulas for the initial sequent and the relation representation mentioned above (cf. Definition 5.30 on p. 154 and Example 5.29 on p. 153):

- $\langle \{?\mathit{var}(t)\}, \emptyset \Vdash \mathit{goal}[v, t] \rangle$
- $\langle \{?\mathit{apply}(t)\}, \mathit{IH} \Vdash \mathit{goal}[v, t] \rangle$
  where $\mathit{IH} := \{\mathit{forall.list}(\lambda s : \mathit{term}[@\,V, @F].\ \mathit{goal}[v', s],\ \mathit{args}(t))\}$

Simplification of the induction hypothesis. The induction hypothesis is simplified by symbolic evaluation [73]:

$$
\begin{array}{l}
\mathit{forall.list}(\lambda s : \mathit{term}[@\,V, @F]. \\
\qquad\qquad \mathit{if}\{?\mathit{var}(v'), \mathit{if}\{\mathit{groundterm}(s), \neg\, \mathit{subterm}(v', s), \mathit{true}\}, \mathit{true}\}, \\
\qquad\qquad \mathit{args}(t))
\end{array}
$$

$\Rightarrow_P^*$ (see Example 5.35 on p. 164)

$$
\begin{array}{l}
\quad \mathit{if}\{?\mathit{var}(v'), \\
\quad\quad \mathit{forall.list}(\lambda s : \mathit{term}[@\,V, @F]. \\
\qquad\qquad\qquad \mathit{if}\{\mathit{groundterm}(s), \neg\, \mathit{subterm}(v', s), \mathit{true}\}, \\
\qquad\qquad\qquad \mathit{args}(t)), \\
\quad\quad \mathit{forall.list}(\lambda s : \mathit{term}[@\,V, @F].\ \mathit{true},\ \mathit{args}(t))\}
\end{array}
$$

$\Rightarrow_P$ *(Evaluate else-part), (Trivial forall-call)*

$$
\begin{array}{l}
\quad \mathit{if}\{?\mathit{var}(v'), \\
\quad\quad \mathit{forall.list}(\lambda s : \mathit{term}[@\,V, @F]. \\
\qquad\qquad\qquad \mathit{if}\{\mathit{groundterm}(s), \neg\, \mathit{subterm}(v', s), \mathit{true}\}, \\
\qquad\qquad\qquad \mathit{args}(t)), \\
\quad\quad \mathit{true}\}
\end{array}
$$

Thus the induction hypothesis states that for each variable v' and each $s \in \mathit{args}(t)$, v' is not a subterm of s if s is a ground term.

5.5.2 Proof of the Base Case

The sequent of the base case is given by $\langle H, IH \Vdash \mathit{goal}[v, t]\rangle$ with:

- $H := \{?\mathit{var}(t)\}$
- $IH := \emptyset$
- $\mathit{goal}[v, t] := \mathit{if}\{?\mathit{var}(v), \mathit{if}\{\mathit{groundterm}(t), \neg\, \mathit{subterm}(v, t), \mathit{true}\}, \mathit{true}\}$

Since $\mathit{groundterm}(t) \Rightarrow_{P,H}^* \mathit{false}$, $\mathit{goal}[v, t]$ is evaluated as follows:

$$\quad \mathit{if}\{?\mathit{var}(v), \mathit{if}\{\mathit{groundterm}(t), \neg\, \mathit{subterm}(v, t), \mathit{true}\}, \mathit{true}\}$$

$\Rightarrow_{P,H}^*$ *(Evaluate then-part), (Evaluate condition)*

$$\quad \mathit{if}\{?\mathit{var}(v), \mathit{if}\{\mathit{false}, \neg\, \mathit{subterm}(v, t), \mathit{true}\}, \mathit{true}\}$$

$\Rightarrow_{P,H}^*$ *(Evaluate then-part), (Keep else-part)*

$$\quad \mathit{if}\{?\mathit{var}(v), \mathit{true}, \mathit{true}\}$$

$\Rightarrow_{P,H}^*$ *(Skip condition)*

$$\quad \mathit{true}$$

Thus the base case is proved.

5.5.3 Proof of the Step Case

The sequent of the step case is given by $\langle H, IH \Vdash goal[v,t]\rangle$ with:

- $H := \{?apply(t)\}$
- $IH :=$

$$\begin{array}{l}\{if\{?var(v'),\\ \quad forall.list(\lambda s : term[@V, @F].\\ \qquad\qquad if\{groundterm(s), \neg\, subterm(v', s), true\},\\ \qquad\qquad args(t)),\\ \quad true\}\}\end{array}$$

- $goal[v,t] := if\{?var(v), if\{groundterm(t), \neg\, subterm(v,t), true\}, true\}$

Since

$$groundterm(t) \Rightarrow^*_{P,H} every(groundterm, args(t))$$

and

$$subterm(v,t) \Rightarrow^*_{P,H} some(\lambda s : term[@V, @F].\, subterm(v,s),\ args(t))\,,$$

$goal[v,t]$ is evaluated as follows:

$$if\{?var(v), if\{groundterm(t), \neg\, subterm(v,t), true\}, true\}$$

$\Rightarrow^*_{P,H}$ *(Evaluate then-part), (Evaluate then-part)*

$$\begin{array}{l}if\{?var(v),\\ \quad if\{every(groundterm, args(t)),\\ \qquad \neg\, some(\lambda s : term[@V, @F].\, subterm(v,s),\ args(t)),\\ \qquad true\},\\ \quad true\}\end{array}$$

$\Rightarrow_{P,H}$ *(Evaluate then-part), (Evaluate then-part), (Negative assumption)*[10]

$$\begin{array}{l}if\{?var(v),\\ \quad if\{every(groundterm, args(t)),\\ \qquad \neg\, if\{\neg\, every(groundterm, args(t)),\\ \qquad\qquad true,\\ \qquad\qquad \neg\, forall.list(L, args(t))\},\\ \qquad true\},\\ \quad true\}\end{array}$$

[10]The assumption is given by lemma (5.23). The required substitution is $\sigma := \{p/groundterm,\ q/\lambda s : term[@V, @F].\, subterm(v,s),\ k/args(t)\}$. We write L for the λ-expression $\lambda s : term[@V, @F].\, if\{groundterm(s), \neg\, subterm(v,s), true\}$.

$\Rightarrow^*_{P,H}$ *(Evaluate then-part), (Evaluate then-part), (Affirmative hypothesis)*

$$if\{?var(v),\ if\{every(groundterm, args(t)),\ \neg\, if\{false,\ true,\ \neg\, forall.list(L, args(t))\},\ true\}\ true\}$$

$\Rightarrow^*_{P,H}$ *(Evaluate then-part), (Evaluate then-part)*

$$if\{?var(v),\ if\{every(groundterm, args(t)),\ forall.list(L, args(t)),\ true\}\ true\}$$

$\Rightarrow_{P,H}$ *(Evaluate then-part), (Evaluate then-part), (Affirmative assumption)*[11]

$$if\{?var(v), if\{every(groundterm, args(t)), ?var(v), true\}, true\}$$

$\Rightarrow_{P,H}$ *(Evaluate then-part), (Evaluate then-part), (Affirmative hypothesis)*

$$if\{?var(v), if\{every(groundterm, args(t)), true, true\}, true\}$$

$\Rightarrow_{P,H}$ *(Evaluate then-part), (Skip condition)*

$$if\{?var(v), true, true\}$$

$\Rightarrow_{P,H}$ *(Skip condition)*

$$true$$

Thus the step case is proved.

[11] The assumption is given by the induction hypothesis. The substitution is $\sigma := \{v'/v\}$.

5.6 Summary

To prove theorems about procedures by induction, we instantiate the principle of *well-founded induction* for two special cases:

- For each data structure $str[@A_1, \ldots, @A_k]$, we synthesize a relation representation $\mathcal{R}_{str}$. Well-founded induction wrt. $\mathcal{R}_{str}$ is the principle of *structural induction.*

- For each terminating procedure *proc*, we synthesize a relation representation $\mathcal{R}_{proc}$. Well-founded induction wrt. $\mathcal{R}_{proc}$ is the principle of *induction wrt. the recursion structure of proc*.

  Using information from termination analysis we optimize $\mathcal{R}_{proc}$ [85, 89] so that the resulting induction formulas can be proved more easily: Base cases may become step cases so that an induction hypothesis is available for the proof. Furthermore, induction hypotheses may become more general so they are more widely applicable.

  If a procedure is defined by second-order recursion, the corresponding induction formulas use quantification procedures to express the finite quantification over the $\mathcal{R}_{proc}$-predecessors in the induction hypotheses. We developed a method to optimize these quantification procedures as well to extend the range of the quantification. Again, this strengthens the induction hypotheses.

For the proofs of the base and step cases of an induction, we provide rules for symbolic evaluation that handle λ-expressions and calls of second-order procedures. An example proof shows how a property that involves two procedures with second-order recursion can be verified using these techniques.

Chapter 6

Related Work

Both ACL2 [2, 58] and Isabelle/HOL [5, 66] (using the IsaPlanner [41]) provide heuristic support to prove termination of procedures, to synthesize and select induction axioms, and to prove the base and step cases of an inductive proof. We discuss ACL2 and Isabelle/HOL in the following sections.

In PVS [7, 67, 69, 77] it is always the user's responsibility to provide a measure function in order to prove termination of a procedure. To prove a formula by induction, the user needs to specify the induction variable and the induction axiom (omitting the specification of an induction axiom leads to structural induction). Thus there is no heuristic support in PVS that is related to our approach.

Termination analysis itself is an area of extensive research. Therefore we also discuss related work that focuses on just this aspect of program analysis. In particular, we describe some preliminary work that aims at automating termination analysis in Coq.

6.1 ACL2

Although ACL2 is based on first-order logic, it is possible to define second-order-like procedures. Such procedures are no "real" second-order procedures, but instead call an undefined first-order function symbol. This undefined function symbol can be "functionally instantiated" with an existing terminating first-order function [32].

For instance, procedure *map* (cf. Figure 1.3 on p. 6) could be defined as a first-order procedure that calls an undefined function symbol $f : @A \to @B$:

$$
\begin{array}{l}
\texttt{procedure}\ map(k : list[@A]) : list[@B] <= \\
\mathit{if}\ ?\varepsilon(k) \\
\quad \mathit{then}\ \varepsilon \\
\quad \mathit{else}\ \ f(hd(k)) :: map(f, tl(k)) \\
\mathit{end}
\end{array}
$$

```
procedure groundterm(t : term[@V, @F]) : bool <=
case t of
  var   : false,
  apply : groundtermlist(args(t))
end

procedure groundtermlist(k : list[term[@V, @F]]) : bool <=
if ?ε(k)
  then true
  else if groundterm(hd(k))
         then groundtermlist(tl(k))
         else false
end    end
```

Figure 6.1: Defining procedure *groundterm* using mutual recursion

By instantiating *map* with $\{f/len\}$, we get procedure *get.lengths* of Figure 1.2 (p. 5). Instantiation of *map* with $\{f/sort\}$ yields procedure *sort.lists*.

Functional instantiation does *not* allow the user to define procedures by second-order recursion, however: Suppose that we want to implement procedure *groundterm* (cf. Figure 1.5 on p. 9). We implement procedure *every* with an undefined function symbol $p : @A \rightarrow bool$:

```
procedure every(k : list[@A]) : bool <=
if ?ε(k)
  then true
  else if p(hd(k))
         then every(p, tl(k))
         else false
end    end
```

Before we can implement *groundterm*, we need to functionally instantiate *every* by $\{p/groundterm\}$. But this instantiation of *every* cannot be defined, because procedure *groundterm* does not exist yet. This circular dependency makes second-order recursion impossible in ACL2.

Although the instantiation of *every* with $\{p/groundterm\}$ cannot be defined in ACL2, we can directly define a procedure *groundtermlist* that checks if each term in a list is a ground term. The resulting procedures shown in Figure 6.1 are defined by *mutual recursion*, which is supported by ACL2.

Using mutual recursion instead of second-order recursion has several drawbacks:

1. The concept "iteration over lists" needs to be coded over and over again: We get procedures *groundtermlist* (to check if *each* term in a

list is a ground term), *subtermlist* (to check if *each* term in a list is a subterm of *some* term in another list), *varcntlist* (to count all variables in a list of terms), ... Code duplication is a bad programming style and should be avoided.

2. As a rule of thumb, the more procedures a program contains the more lemmas are required to prove a property of the program. This makes program verification much more difficult: Additional lemmas need to be proved for auxiliary procedures such as *groundtermlist*.

3. Proving properties of procedures that are defined by mutual recursion is often difficult: Usually one cannot consider these procedures individually and prove a property of just a single procedure. Instead it is necessary in general to prove a more complex property about all procedures that are involved in the mutually recursive definition.

   For instance, a property of *groundterm* would have to be generalized to a conjunction of properties of *groundterm* and *groundtermlist*, which needs to be proved simultaneously. The user manual of ACL2[1] gives a simple example proof (namely to show that a procedure *variables* : $term[@V, @F] \rightarrow list[@V]$ indeed yields a list of variables) which is already so complex that the induction scheme for the conjunctive statement needs to be supplied as a hint to ACL2.[2]

   In [65], Moore therefore concludes:

   > "ACL2 supports mutual recursion. But mutual recursion can be awkward when dealing with induction. To prove an inductive theorem about one function in a clique of mutually recursive functions, one must often prove a conjunction of related theorems about the other functions of the clique. While ACL2 can often manage the proofs, the user must state the conjunction of theorems in order for the conjecture to be strong enough to be provable. We found this often inconvenient, especially in the early going when nothing but the definitions of the functions are available."

Circumventing second-order recursion and mutual recursion. As procedure *groundterm* does not contain a direct recursive call in Figure 6.1, the call $groundterm(hd(k))$ in procedure *groundtermlist* can be unfolded

[1] `http://www.cs.utexas.edu/users/moore/acl2/v3-4/acl2-doc.html`

[2] For certain cases (e. g., if the original theorem involves just a single procedure of a clique of mutually recursive procedures) Boulton and Slind [31] describe how a so-called *multi-predicate* induction scheme can be derived automatically from the mutually recursive procedure definitions. However, the main problem with proofs about mutually recursive procedures persists: The resulting proof obligations are complex conjunctions about *all* procedures of the clique of mutual recursive procedures.

(by replacing it with the instantiated body of *groundterm*) so that procedure *groundtermlist* is no longer defined by mutual recursion. The result is shown in Figure 6.2. However, we cannot prove anything about *groundterm* without proving auxiliary lemmas about *groundtermlist*. Thus for each lemma about *groundterm* we need to state and prove a corresponding lemma about *groundtermlist*.

The difficulty with this approach becomes even more apparent if we consider a slightly more complex procedure than *groundterm*. Figure 6.2 shows the result of implementing procedure *subterm* (cf. Figure 1.5 on p. 9) without second-order recursion and mutual recursion. Procedure *subtermlist* returns *true* iff for each term r in list k there exists a term t in list l such that r is a subterm of t.[3] While the implementation of procedure *subterm* in Figure 1.5 exactly corresponds to the usual definition of the subterm relation, the meaning of *subtermlist* is hidden within the code that iterates over the lists.

Summing up, our support of second-order recursion facilitates a natural implementation of many algorithms *and* simplifies the proofs of properties of the implementation.

6.2 Isabelle/HOL

Since Isabelle/HOL is based on higher-order logic, which is *not* a programming language, it cannot find out which subterms of a term t "need to be evaluated" to evaluate t. Thus termination of procedures that use second-order recursion (e. g., *groundterm*, *subterm*, *varcount*) cannot be proved without intervention by the user. Furthermore, induction axioms cannot be optimized.

Termination analysis. A termination proof of procedure

$$
\begin{array}{l}
\texttt{procedure}\ \mathit{groundterm}(t : \mathit{term}[@\,V, @F]) : \mathit{bool} <= \\
\mathit{case}\ t\ \mathit{of} \\
\quad \mathit{var}\ \ : \mathit{false}, \\
\quad \mathit{apply} : \mathit{every}(\lambda s : \mathit{term}[@\,V, @F].\, \mathit{groundterm}(s),\ \mathit{args}(t)) \\
\mathit{end}
\end{array}
$$

yields the unprovable termination condition that there is a well-founded relation $\succ$ on $\mathit{term}[@\,V, @F]$ with $t \succ s$ for all $t, s : \mathit{term}[@\,V, @F]$, see [79]. This is because Isabelle cannot recognize in which cases evaluation of procedure call $\mathit{groundterm}(s)$ is required.

[3] We need to transform both parameters of *subterm* into lists of terms to be able to state transitivity of *subtermlist*: $\forall k_1, k_2, k_3 : \mathit{list}[\mathit{term}[@\,V, @F]].\ \mathit{subtermlist}(k_1, k_2) \wedge \mathit{subtermlist}(k_2, k_3) \rightarrow \mathit{subtermlist}(k_1, k_3)$.

```
procedure groundterm(t : term[@V, @F]) : bool <=
case t of
  var   : false,
  apply : groundtermlist(args(t))
end

procedure groundtermlist(k : list[term[@V, @F]]) : bool <=
if ?ε(k)
  then true
  else case hd(k) of
         var   : false,
         apply : if groundtermlist(args(hd(k)))
                   then groundtermlist(tl(k))
                   else false
end    end       end

procedure subterm.of.some(r : term[@V, @F],
                          l : list[term[@V, @F]]) : bool <=
if ?ε(l)
  then false
  else if r = hd(l)
         then true
         else case hd(l) of
                var   : subterm.of.some(r, tl(l))
                apply : if subterm.of.some(r, args(hd(l)))
                          then true
                          else subterm.of.some(r, tl(l))
end    end    end       end

procedure subtermlist(k, l : list[term[@V, @F]]) : bool <=
if ?ε(k)
  then true
  else if subterm.of.some(hd(k), l)
         then subtermlist(tl(k), l)
         else false
end    end
```

Figure 6.2: Defining procedures *groundterm* and *subterm* without second-order recursion and mutual recursion

To solve this problem, the user needs to prove a *congruence theorem*

$$\forall k, k' : list[@A],\ p, p' : @A \to bool.$$
$$k = k' \wedge \big(\forall x : @A.\ x \in k \to p(x) = p'(x)\big) \to every(p, k) = every(p', k')$$

about procedure *every* [62, 66, 79]. As soon as the user *explicitly tags* this theorem as a *congruence rule*, Isabelle assumes that evaluation of $every(p, k)$ only requires evaluation of $p(x)$ for all $x \in k$ (where "$x \in k$" comes from the congruence rule). Using this hint, Isabelle generates the provable termination condition that there is a well-founded relation $\succ$ on $term[@V, @F]$ with

$$\forall t, s : term[@V, @F].\ s \in args(t) \to t \succ s\,.$$

Isabelle proves this termination condition by using the *structural size* relation on terms, defined by $t \succ s$ iff $\#(t, \epsilon) > \#(s, \epsilon)$ [62].

Induction axioms. Induction axioms[4] are derived from the termination proofs and thus are directly influenced by the congruence rules that were defined by the user. We present the induction axiom that Isabelle computes for procedure *groundterm* by displaying the induction formulas for a proof of $\forall t : term[@V, @F].\ goal[t]$ in order to facilitate a comparison with Example 5.29 (p. 153):

- $?var(t) \to goal[t]$
- $?apply(t) \wedge \big(\forall s : term[@V, @F].\ s \in args(t) \to goal[s]\big) \to goal[t]$

These induction formulas are equal to the induction formulas that our approach generates, apart from the different notation of the finite quantification (which we discuss in Chapter 7). This induction axiom is optimal; the generalization (which our approach carries out automatically) has been supplied by the user via the congruence rule about *every* (which states that evaluation of $p(x)$ for *all* $x \in k$ is generally required).

However, even the "best" congruence rules do not guarantee that the induction axioms generated by Isabelle are optimal. An example given in [62] by Krauss shows that the use of congruence rules may easily lead to suboptimal induction axioms: The Isabelle definition

$$foo\ n = (n = 0 \vee foo\ (n - 1))$$

corresponds to the procedure definition

[4] Actually, *induction theorems* are generated, because well-foundedness of the underlying relation is proved within Isabelle's higher-order logic.

```
procedure foo(n : ℕ) : bool <=
if ?0(n)
  then true
  else foo(⁻(n))
end
```

in our notation (cf. Table 2.2 on p. 40). According to the semantics of our programming language, *foo* clearly terminates. However, termination of *foo* according to the Isabelle definition depends on the operational semantics of $\vee$: If $a \vee b$ always evaluates a and b, then the Isabelle definition does *not* denote a terminating function, because evaluation of $foo(0)$ requires evaluation of $foo(0)$, as $0 - 1$ evaluates to 0. But if $a \vee b$ evaluates a first and only evaluates b if a evaluates to *false*, then *foo* is also considered as a total function by Isabelle.

In order to reflect the common semantics of programming languages (e. g., ML or Java) that evaluate b only if a evaluates to *false*, the user adds a congruence rule

$$\forall a, b, a', b' : bool.$$
$$[a = a' \wedge (\neg\, a' \rightarrow b = b')] \rightarrow (a \vee b) = (a' \vee b')\,.$$

Using this congruence rule, termination of *foo* can be proved. Unfortunately, this congruence rule leads to a suboptimal induction axiom for another procedure. The Isabelle definition

$$\begin{array}{lcl} contains.zero\ [\,] & = & False \\ contains.zero\ (x : xs) & = & (x = 0) \vee contains.zero\ xs \end{array}$$

corresponds to the procedure definition

```
procedure contains.zero(k : list[ℕ]) : bool <=
if ?ε(k)
  then false
  else if ?0(hd(k))
         then true
         else contains.zero(tl(k))
       end
end
```

in our notation. The induction axiom synthesized by Isabelle yields the following induction formulas for a proof of $\forall k : list[\mathbb{N}].\ goal[k]$:

- $goal[\varepsilon]$
- $\neg\, ?\varepsilon(k) \wedge \big(\neg\, ?0(hd(k)) \rightarrow goal[tl(k)]\big) \rightarrow goal[k]$

In the step case, we can only use $goal[tl(k)]$ if we can show $\neg\, ?\mathtt{0}(hd(k))$, which makes this induction axiom significantly weaker than the common list induction axiom. In contrast, our approach identifies $?\mathtt{0}(hd(k))$ as unnecessary condition (for the termination proof) and thus generates the optimal list induction axiom given by the induction formulas

- $goal[\varepsilon]$
- $\neg\, ?\varepsilon(k) \wedge goal[tl(k)] \rightarrow goal[k]\,.$

Consequently, Krauss concludes that "there is no 'best' or 'complete' set of congruence rules". In other words, it is impossible to axiomatize the evaluation order of a programming language via congruence rules in such a way that (i) termination can be (easily) proved and (ii) the induction axioms are sufficiently general to facilitate inductive proofs of formulas.

6.3 Stand-Alone Termination Provers

Stand-alone termination provers solely address the question whether a given program terminates or not. Some tools are quite powerful and highly automated. However, state-of-the-art theorem provers do not use these techniques, because no methods are known yet that facilitate the synthesis of optimized induction axioms from terminating procedures.

6.3.1 Dependency Pairs

The method of *dependency pairs* [17, 45] is a technique to show termination of term rewrite systems. Aoto and Yamada [16] use this method to prove termination of simply typed term rewrite systems that contain rewrite rules for procedures such as *filter* and *qsort*.

Giesl et al. [44] show that the dependency pair method can also be used to prove termination of programs written in a real programming language. Their approach considers Haskell programs and has been implemented in AProVE [43], which is a tool for automated termination analysis. AProVE generates a *termination graph* that approximates the $\rhd$-relation for procedure calls. This graph is transformed into dependency pairs problems, which are then tackled by existing techniques from term rewriting.

The approach by Giesl et al. assumes that a Haskell program does not contain λ-expressions. Having to translate procedures into a λ-free form (even if this is done automatically) brings about two difficulties: Firstly, this is another obstacle to the synthesis of induction axioms, because the termination proof for the λ-free program needs to be translated back to the original program (because an induction axiom for the *original* program is required, not for the translated program). Secondly, any kind of internal

program transformation is disadvantageous when the user is supposed to interactively solve subproblems, because these subproblems involve statements about a transformed program that the user does not know about.

An advantage of AProVE is that it can also prove termination of a procedure if the parameter does *not* get structurally smaller, but is incremented up to a certain bound [46].

Giesl's approach cannot prove termination of procedures that require an inductive proof of a termination hypothesis; e. g., for the recursive call $minsort(remove(min(k), k))$ in an implementation of Minimum Sort, one needs to prove $\forall k : list[\mathbb{N}].\ k \neq \varepsilon \rightarrow min(k) \in k$ by induction to verify that the list gets smaller [86].

6.3.2 Higher-Order Recursive Path Orderings

Jouannaud and Rubio [55] developed a termination proof technique via *higher-order recursive path orderings* (HORPO). This technique is intended to be a step towards the automation of termination proofs for higher-order rewrite systems.

Their approach is oversized for our setting, as it goes beyond the analysis of functional programs that we consider. Some parts of this technique have been automated, but the user still needs to supply extra information to start termination analysis: a well-founded quasi-ordering on types, a well-founded quasi-ordering on function symbols (called *precedence*), and a status ("multiset" or "lexicographic") for each type constructor and each function symbol.

6.3.3 Size-Change Termination

Sereni [75] extends the *size-change termination* framework beyond size measures for base types. However, his approach fails on certain termination problems that build on argument-bounded procedures: It cannot prove termination of procedure *msort* (cf. Example 4.46 on p. 116), because it does not recognize argument-boundedness of *split*. Similarly, it cannot prove termination of procedure *qsort* (cf. Example 4.44 on p. 115), because it does not recognize argument-boundedness of *filter* [74].

6.3.4 Sized Types and Dependent Types

Termination of a procedure can be shown by *typing* within a type system that allows to annotate types with size information [9, 64]. Suppose that $list_n[@A]$ denotes the type of lists of length $\leq n$. Termination of procedure *len* in Figure 1.1 (p. 4) can be shown as follows:

Example 6.1. Let term variable k be of type $list_{n+1}[@A]$ and assume that *len* is a function symbol of type $list_n[@A] \to \mathbb{N}$. If the body

$$\mathit{if}\ ?\varepsilon(k)\ \mathit{then}\ \mathtt{0}\ \mathit{else}\ {}^{+}(\mathit{len}(\mathit{tl}(k))) \tag{6.1}$$

of procedure *len* can by typed, then *len* terminates (i. e., *len* computes a total function). Indeed, (6.1) can be typed, because *tl* has type $list_{n+1}[@A] \to list_n[@A]$. Thus the type of selector *tl* encodes that *tl* is argument-bounded (or *size-preserving*, as it is called in [9]). ♢

Obviously, inferring useful size annotations is a major challenge. Chin and Khoo [39] study the size inference problem and provide an algorithm that generates formulae of Presburger arithmetic that guarantees termination of a procedure.

Barthe et al. [22, 23, 24] have developed a prototype implementation of a type checker and size inference algorithm that is able to show both termination and argument-boundedness of Quicksort. While our method automatically proves termination of Quicksort as well, it cannot show argument-boundness of that procedure. However, termination of Mergesort (cf. Figure 4.8 on p. 117) cannot be proved using the approach by Barthe et al.: While the type systems recognize argument-boundedness of procedure *split* (cf. Figure 4.2 on p. 95), they do not recognize the cases when *split* returns lists that are *strictly* smaller than the input list. The concept of difference functions precisely captures the behavior of *split* so our approach easily shows termination of Mergesort.

Blanqui [30] describes a termination criterion for higher-order conditional rewriting that uses more precise size information (e. g., that *split* returns two lists whose sizes add up to the size of the input list). In contrast to Barthe et al., he assumes that the types of function symbols (including size information) are already given and merely checks that they imply termination. Neither the approach by Barthe et al. nor the one by Blanqui are ready yet for integration into Coq's calculus of inductive constructions [29]; this future work (involving several extensions) is considered as long-term goal [23, 30].

Work by Abel [8] and Hughes et al. [53] considers "non-standard type systems" that can also be used to ensure *productivity* of algorithms that work on coinductive data structures such as infinite streams.

In neither of the aforementioned approaches of type-based termination analysis the problem of second-order recursion has been considered. Also, the tool described in [21] that generates induction axioms from procedure definitions in Coq fails in these cases.

Xi [100] considers termination analysis of programs with general recursion (including second-order recursion and mutual recursion) via *dependent types*. His approach verifies termination using type annotations given by the

programmer. Of course, one could imagine a heuristic that guesses type annotations to express argument-boundedness of procedures; however, proving termination of Mergesort would fail for the same reason as in the case of sized types (see above).

Chapter 7

Evaluation

Our approach to verify functional programs with second-order recursion has been implemented in ✓eriFun [1, 99]. In order to show that the approach is feasible, we evaluated it on several examples. The details of these examples can be found in Appendix A. The case studies comprise

- a verification of an implementation of Quicksort that uses second-order procedures,
- common operations on terms (*groundterm*, *subterm*, *termsize*, *variables*, *wellformed*, *apply.subst*) and the verification of several properties of these operations,
- an interpreter of a subset of Pure Lisp, and
- flattening a tree of variadic degree into a list along with a proof that the list contains the same elements as the tree.

Organization of this chapter. Section 7.1 describes our experiments with termination analysis. In Section 7.2 we show how our approach performs wrt. inductive proofs. We also describe an alternative to quantification procedures that might be useful when integrating our approach into other theorem provers.

7.1 Termination Analysis

We evaluated our approach to automated termination analysis on the standard examples of procedures with second-order recursion that are considered in the literature [61, 62, 66, 69, 79] and the examples given in this thesis (including Appendix A). In total, our set of examples includes

- 21 procedures that are defined by second-order recursion,
- more than 25 procedures that do not use second-order recursion.

```
procedure zero(n : ℕ) : ℕ <=
funpow(n, zero, 0)

procedure countnodes(k : list[bin.tree[@A]]) : ℕ <=
if ?ε(k)
  then 0
  else if ?tip(hd(k))
         then 0
         else ⁺(countnodes(left(hd(k)) :: right(hd(k)) :: tl(k)))
       end
end
```

Figure 7.1: Examples where automated termination analysis fails

Termination of all these procedures is proved automatically and yields the optimal induction axiom for each procedure. On an Intel Core2duo CPU (2.4 GHz) the single-threaded implementation takes less than 10 seconds for each procedure.

This time could be reduced by stopping termination analysis as soon as the first termination proof succeeds. However, then the implementation would miss some induction axioms. For instance, procedure "!!" (cf. Figure 2.2 on p. 38) terminates for two reasons: List parameter k decreases in recursive calls *and* index n decreases in recursive calls. If termination analysis stopped when termination of "!!" wrt. parameter k was proved, then we would miss the induction axiom wrt. parameter n.

Figure 7.1 shows two procedures that our approach cannot prove terminating: Procedure *zero* is given in [62] as an example that cannot be proved terminating by Isabelle either (see Figure 2.5 on p. 47 for the definition of procedure *funpow*). It returns 0 for all n. The termination proof fails, because we need to know about the semantics of *zero* to prove termination of *zero*. However, a theorem about the semantics of *zero* can only be proved *after* having proved termination of *zero*.

Procedure *countnodes* [50] cannot be proved terminating by our approach, because the estimation calculus (cf. Figure 4.4 on p. 103) is unable to handle arithmetic. For the termination proof we would need to show

$$\#(k, \mathbf{1}) > \#(\mathit{left}(\mathit{hd}(k)) :: \mathit{right}(\mathit{hd}(k)) :: \mathit{tl}(k), \mathbf{1})\,,$$

which is equivalent to

$$\#(k, \mathbf{1}) > \#(\mathit{left}(\mathit{hd}(k)), \epsilon) + \#(\mathit{right}(\mathit{hd}(k)), \epsilon) + \#(\mathit{tl}(k), \mathbf{1})\,.$$

The estimation calculus can only show $\#(k, \mathbf{1}) > n$ for each summand n. Thus the recursive call would have to be rewritten as

$$\begin{aligned} &{}^{+}(\mathit{countnodes}(\mathit{left}(\mathit{hd}(k)) :: \varepsilon) + \\ &\quad \mathit{countnodes}(\mathit{right}(\mathit{hd}(k)) :: \varepsilon) + \\ &\quad \mathit{countnodes}(\mathit{tl}(k))) \end{aligned}$$

so that our approach can prove termination of *countnodes*.

Procedure *countnodes* is given by Gries in [50] to illustrate how a recursively defined procedure can be transformed into an iterative procedure by explicitly maintaining a stack (Gries uses a *set* for the stack, whereas we use *lists*). Of course, the original recursive definition is easily proved terminating by our approach; it is defined on a single binary tree $t : \mathit{bin.tree}[@A]$. So whenever one needs to count the nodes in a list $k : \mathit{list}[\mathit{bin.tree}[@A]]$ of binary trees, one can lift this procedure that operates on a single binary tree to lists k by $\mathit{listsum}(\mathit{map}(\mathit{countnodes}, k))$ (see Figure 7.3 on p. 190 for the definition of procedure *listsum*).

Despite this limitation, our approach constitutes a significant improvement regarding automated theorem proving: Unlike in Isabelle, termination of all procedures mentioned above (except the artificial *zero* example and *countnodes*) is proved automatically, whereas user interaction is needed in Isabelle (by supplying congruence rules) in case of second-order recursion. In addition to all procedures that the original approach via argument-bounded functions [86, 96] is able to prove terminating, our approach also improves termination analysis in the first-order case by proving termination of procedures *msort* and *qsort* automatically.

7.2 Inductive Theorem Proving

In order to demonstrate that the high degree of automation that has been obtained for the verification of first-order programs (e. g., in verification tools such as ACL2 and ✓`eriFun`) can be maintained for the verification of second-order programs, we proved several theorems with an implementation of our approach in ✓`eriFun`.

Quicksort. We proved that procedure *qsort* computes an ordered permutation of the input list. The implementation uses the second-order procedure *filter*. Second-order recursion is *not* involved in this case study. Thus a very similar case study could be carried out with the earlier version of ✓`eriFun` that was limited to first-order procedures (instead of procedure *filter*, we used two procedures *smaller* and *larger* that select the list elements that are smaller or larger than the pivot element, respectively).

The results for Quicksort are:

- The second-order implementation requires only 6 user-defined procedures instead of 9 procedures in the first-order implementation: Procedure *filter* generalizes procedure *smaller* and *larger*, and two procedures to check if a number is an upper or lower bound of a list of numbers are generalized by the system-generated procedure *forall.list*.
- The second-order implementation requires only 9 auxiliary lemmas to prove the 2 main lemmas. In contrast, the first-order implementation requires 14 auxiliary lemmas.
- Apart from defining the auxiliary lemmas, no user interaction is necessary. In particular, all induction and termination proofs were obtained automatically by the system.

Term operations. Several preliminary experiments revealed that one frequently needs auxiliary lemmas that relate procedures such as *forall.list*, *every*, and *some* to each other (cf. the example proof in Section 5.5).

This is not surprising, because generally for each auxiliary procedure one needs at least one auxiliary lemma that states a certain property of this procedure. Since *forall.list*, *every*, and *some* more or less compute the same function (up to negation), we chose to implement the procedures on terms (e. g., *groundterm* and *subterm*) using just a single quantification procedure, namely *forall.list*.[1]

The statistics for this case study are:

- Altogether 30 lemmas were proved. Out of the 11 main lemmas, 10 lemmas used an induction axiom that was derived from a procedure with second-order recursion.
- All of the 29 inductive proofs were successfully obtained using the induction axiom automatically chosen by the system. (Reflexivity of *subterm* can be proved without induction.)
- Only 2 user interactions were needed:
  - For the proof of "subterm(r, t) $\rightarrow$ termsize(r) $\leq$ termsize(t)", lemma "$x \neq 0 \wedge x \leq y \rightarrow {}^{-}(x) \leq y$" was applied interactively.
  - For the proof of "termsize(t) $\leq$ termsize(σ(t))", lemma "map(g, map(f, k)) = map(g $\circ$ f, k)" was applied interactively.

We discuss some further observations in the following subsections.

[1] The lemmas listed in Appendix A.2 can of course also be verified without such a unified implementation. However, in practice one should generally limit the number of auxiliary procedures to simplify the proofs.

```
structure sexpr <=
  nil,
  lispsymbol(lname : predefinedSymbol),
  usersymbol(uname : ℕ),
  number(value : ℕ),
  cons(car : sexpr, cdr : sexpr)

structure result[@A] <=
  ⊥,
  def(val : @A)
```

Figure 7.2: Data structure definitions *sexpr* and *result*[@*A*]

Lisp interpreter. Boyer and Moore [34] describe a machine-checked proof of the undecidability of the halting problem. They use a subset of *Pure Lisp* as a model of computation. The semantics of Lisp programs is defined by an interpreter

$$\texttt{procedure}\ eval(expr, va, fa : sexpr,\ n : \mathbb{N}) : result[sexpr]\,.$$

The input of the interpreter is an s-expression *expr* to be evaluated (cf. Figure 7.2 and Appendix A.3), an assignment *va* of values to variable symbols, an assignment *fa* of function definitions to function symbols, and a resource limit n (indicating the maximum depth of function calls to ensure termination of procedure *eval*). The assignments *va* and *fa* are represented as s-expressions and model association lists, i. e., lists of pairs (key, value). Both variable symbols and function symbols are represented by s-expressions of the form $usersymbol(n)$. The interpreter either returns $def(r)$ for result $r : sexpr$ of the evaluation or $\bot$ if resource limit n does not suffice to evaluate *expr*.

In order to evaluate a function call $f(t_1, \ldots, t_n)$, the call-by-value interpreter *eval* first needs to evaluate the list l of arguments $t_1, \ldots, t_n$. This could be done by a procedure

$$\texttt{procedure}\ evlist(l, va, fa : sexpr,\ n : \mathbb{N}) : result[sexpr]\,,$$

that considers $l : sexpr$ as a list of s-expressions and calls *eval* for each element of this list. However, to avoid mutual recursion of procedures *eval* and *evlist*, Boyer and Moore use a single procedure

$$\texttt{procedure}\ ev(flag : evflag,\ expr, va, fa : sexpr,\ n : \mathbb{N}) : result[sexpr]$$

that is parameterized by a flag to indicate if parameter *expr* is to be considered as a single s-expression or as an s-expression that represents a list of s-expressions to be evaluated.

Instead of merging the concepts *evaluation of an s-expression* ("*eval*") and *iteration over an s-expression* ("*evlist*") into a single procedure *ev*, our implementation in Appendix A.3 uses *second-order recursion* to implement the interpreter:

- Procedure $mapsx(f : sexpr \to result[sexpr],\ x : sexpr)\ :\ result[sexpr]$ considers s-expression x as a list and applies f to each element of x. It returns $\bot$ iff at least one application of f to an element of x returns $\bot$.

- Procedure $eval(expr, va, fa : sexpr, n : \mathbb{N}) : result[sexpr]$ evaluates a single s-expression *expr*. It uses procedure *mapsx* to evaluate the list of arguments of a function call $f(t_1, \ldots, t_n)$ via second-order recursion.

Similarly to [34], we use the following auxiliary procedures:

- Procedure $get(key, alist : sexpr) : sexpr$ returns the value that *key* is associated with by association list *alist* or *nil* if no association exists for *key*.

- Procedure $pairlist(args, vals : sexpr) : sexpr$ transforms two lists (of equal length) into a list of pairs.

- Procedure *apply* (called `APPLY.SUBR` in [34]) interprets the predefined Lisp symbols.

We proved a key lemma about the interpreter that is required to prove the undecidability of the halting problem: If resource limit n suffices to evaluate *expr* to some result r, then evaluation of *expr* with resource limit $n + k$ for arbitrary $k \in \mathbb{N}$ yields the same result r. Our proof requires an auxiliary lemma (called `mapsx(f, x) = mapsx(g, x)`) that needs to be applied interactively in an induction step. It basically states that $mapsx(f, x)$ is equal to $mapsx(g, x)$ if f and g coincide on the elements of list x. (The version in the appendix is slightly more specialized to match the goal term that arises in the inductive proof.) Apart from this single interaction no user interaction is required, so in particular the system always heuristically selected a suitable induction axiom.

This case study shows that our approach also works well for data structures that are *not* by definition nested in another data structure (for instance, data structure $term[@V, @F]$ is nested because selector *args* has return type $list[term[@V, @F]]$, whereas data structure *sexpr* is not nested because no selector of an *sexpr*-constructor has return type $str[sexpr]$ for some type constructor *str*). Thus second-order recursion is supported even if it does not correspond to structural recursion wrt. the nesting of a data structure.

Variadic trees. Based on data structure *tree*[@A] for variadic trees, we implemented a procedure that flattens a variadic tree to a list. The two main lemmas state that the elements of the resulting list are exactly the items of the tree. These main lemmas both required induction wrt. a recursion structure of a procedure defined by second-order recursion. For all 8 lemmas, the correct induction axiom was automatically and correctly selected by the system. Only one lemma was applied interactively.

7.2.1 Benefit of Optimized Induction Axioms

The optimization of relation representations that leads to optimized induction axioms (cf. Section 5.2.3) is advantageous in two respects:

1. The optimization of relation representations facilitates the heuristic selection of a promising induction axiom for a given formula (the heuristic selection of induction axioms is described in [83, 85]).
2. The optimized induction axioms significantly simplify the proofs.

Benefit (1) is illustrated by the example proof in Section 5.5. To prove

> `lemma` variable not subterm of groundterm $<= \forall v, t : term[@\,V, @F].$
> $if\{?var(v), if\{groundterm(t), \neg\, subterm(v, t), true\}, true\}$

we have four obvious options for an inductive proof:

- structural induction on v, because v is a universally quantified variable that occurs twice in the formula
- structural induction on t, because t is a universally quantified variable that occurs twice in the formula
- induction wrt. *groundterm*, because *groundterm* is defined recursively and occurs in the formula
- induction wrt. *subterm*, because *subterm* is defined recursively and occurs in the formula

Without optimized relation representations, these would be four different possibilities and it would be unclear which one is promising for an inductive proof.

By optimizing the relation representations of *groundterm* and *subterm*, we are in a much better position: The optimized relation representations of procedures *groundterm* and *subterm* coincide and furthermore are equal to the relation representation for structural induction wrt. t. Thus there are three "votes" for structural induction on t and one "vote" for structural induction on v. Indeed, structural induction on t is successful (see Section 5.5), whereas structural induction on v would fail.

```
procedure listsum(k : list[ℕ]) : ℕ <=
if ?ε(k)
  then 0
  else hd(k) + listsum(tl(k))
end

procedure varcnt(t : term[@V, @F]) : ℕ <=
case t of
  var   : 1,
  apply : listsum(map(varcount, args(t)))
end

lemma no variables entails groundterm <=
∀t : term[@V, @F]. if{?0(varcnt(t), groundterm(t), true}
```

Figure 7.3: Lemma "?0($varcnt(t)$) → $groundterm(t)$"

To illustrate benefit (2), let us consider a proof of the lemma given in Figure 7.3: If the number of variables that occur in a term t is zero, then t is a ground term.

A proof by structural induction on t requires two auxiliary lemmas that relate *listsum* (used by *varcnt*), *forall.list* (used in the induction hypothesis), and *every* (used by *groundterm*):

```
lemma "listsum(map(f, k))=0 entails f(x)=0" <=
∀k : list[@A], f : @A → ℕ.
if{?0(listsum(map(f, k))), forall.list(λx : @A. ?0(f(x)), k), true}

lemma "[∀x ∈ k. p(x) ⇒ q(x)] ∧ [∀x ∈ k. p(x)] ⇒ [∀x ∈ k. q(x)]" <=
∀k : list[@A], p, q : @A → bool.
if{forall.list(λx : @A. if{p(x), q(x), true}, k)
   if{forall.list(p, k), every(q, k), true},
   true}
```

Both auxiliary lemmas are easy to spot and correspond to reasoning steps that are also required in a proof using pen and paper.

In contrast, a proof by induction wrt. the non-optimized relation representation of procedure *groundterm* (cf. Example 5.20 on p. 144) requires the following lemmas:

```
lemma "listsum(map(f, k))=0 entails f(x)=0 using forall.every" <=
∀k : list[@A], f : @A → ℕ, q : @A → bool.
if{?0(listsum(map(f, k))), forall.every(λx : @A. ?0(f(x)), q, k), true}
```

$$
\begin{array}{l}
\texttt{lemma}\ \text{“establish every(q, k)”} <= \\
\forall k : \mathit{list}[@A],\ p, q : @A \to \mathit{bool}. \\
\mathit{if}\{\mathit{forall.every}(\lambda x : @A.\ \mathit{if}\{p(x), q(x), \mathit{true}\},\ q,\ k), \\
\quad \mathit{if}\{\mathit{forall.every}(p, q, k), \mathit{every}(q, k), \mathit{true}\}, \\
\quad \mathit{true}\}
\end{array}
$$

These lemmas are more difficult to find, because *forall.every* is a quite complex procedure that is used in the induction hypotheses. The first lemma shows that $?0(f(x))$ holds for all elements x of an initial segment of list k; this initial segment consists of those x that *every* calls q with. The second lemma then shows that if $p(x)$ implies $q(x)$ for all x of this initial segment of k and if $p(x)$ holds for all x of this initial segment, then $q(x)$ holds for *all* elements x of k.

7.2.2 Alternative Formulation of Finite Quantification

In Chapter 3 we investigated finite quantification and described the synthesis of quantification procedures *forall.str* and *forall.proc* that are used, for instance, in induction hypotheses. Alternatively, we could as well synthesize *membership procedures*. We sketch this approach in the following and explain why we prefer quantification procedures to membership procedures.

Membership procedures for data structures. For a data structure definition

$$
\begin{array}{l}
\texttt{structure}\ \mathit{str}[@A_1, \ldots, @A_k] <= \\
\quad \mathit{cons}_1(\mathit{sel}_{1,1} : \tau_{1,1}, \ldots, \mathit{sel}_{1,n_1} : \tau_{1,n_1}), \\
\quad \ldots, \\
\quad \mathit{cons}_m(\mathit{sel}_{m,1} : \tau_{m,1}, \ldots, \mathit{sel}_{m,n_m} : \tau_{1,n_m})\ .
\end{array}
$$

we can uniformly synthesize *membership procedures*

$$\texttt{procedure}\ \mathit{member.str}_h(z : @A_h,\ x : \mathit{str}[@A_1, \ldots, @A_k]) : \mathit{bool}$$

for each $h = 1, \ldots, k$. The semantics of these procedures is

$$\mathit{eval}_P(\mathit{member.str}_h(\mathsf{z}, \mathsf{x})) = \mathit{true} \iff \mathsf{z} \in \mathit{Itm}(\mathsf{x}, h)$$

for all values $\mathsf{x}, \mathsf{z} \in \mathbb{V}(P)$.

Membership procedures for second-order procedures. For a procedure with signature

$$\texttt{procedure}\ \mathit{proc}(f : \tau_1 \times \ldots \times \tau_m \to \tau_f,\ x : \tau_x) : \tau_{\mathit{proc}}$$

we can uniformly synthesize the *membership procedure*

$$\texttt{procedure}\ \mathit{member.proc}(z_1 : \tau_1, \ldots, z_m : \tau_m, f : \tau_1 \times \ldots \times \tau_m \to \tau_f, x : \tau_x) : \mathit{bool}$$

that satisfies

$$\mathit{eval}_P(\mathit{member.proc}(\mathsf{z}_1, \ldots, \mathsf{z}_m, \mathsf{f}, \mathsf{x})) = \mathit{true} \iff \mathit{proc}(\mathsf{f}, \mathsf{x}) \rhd_{\mathsf{f}} \mathsf{f}(\mathsf{z}_1, \ldots, \mathsf{z}_m)$$

for all values $\mathsf{z}_1, \ldots, \mathsf{z}_m, \mathsf{f}, \mathsf{x}$.

Using membership procedures in induction hypotheses. Let us again consider the proof of

$$\texttt{lemma}\ \text{variable not subterm of groundterm} <= \forall v, t : \mathit{term}[@V, @F].\ \mathit{if}\{?\mathit{var}(v), \mathit{if}\{\mathit{groundterm}(t), \neg\, \mathit{subterm}(v, t), \mathit{true}\}, \mathit{true}\}\,.$$

In contrast to Section 5.5, we phrase the induction hypothesis of the step case using *member.list* instead of *forall.list*:

$$\mathit{IH} := \big\{\mathit{if}\{?\mathit{var}(v'), \mathit{if}\{\mathit{member.list}(s', \mathit{args}(t)), \mathit{if}\{\mathit{groundterm}(s'), \neg\, \mathit{subterm}(v', s'), \mathit{true}\}, \mathit{true}\}, \mathit{true}\}\big\}$$

Since the induction hypothesis differs from the induction hypothesis we considered in Section 5.5, we need a different lemma to prove the step case. However, this lemma is not a universal formula, because we need to make the implicit universal quantification of s' in the induction hypothesis explicit ("$\forall x : @A$"):

$$\texttt{lemma}\ \text{“}[\forall x \in k.\ p(x) \Rightarrow \neg\, q(x)] \wedge [\forall x \in k.\ p(x)] \Rightarrow [\neg\, \exists x \in k.\ q(x)]\text{”} <= \forall p, q : @A \to \mathit{bool}, k : \mathit{list}[@A].\ [\forall x : @A.\ \mathit{member.list}(x, k) \to (p(x) \to \neg\, q(x))] \to [\mathit{every}(p, k) \to \neg\, \mathit{some}(q, k)]$$

This lemma cannot be expressed in $\mathcal{L}$ due to the implicit existential quantification (cf. the discussion in Section 1.2). Therefore we prefer the use of quantification procedures to membership procedures, which expresses finite quantification more nicely.

However, if the target theorem prover supports arbitrarily nested formulas, our approach can easily be adapted to use membership procedures instead of quantification procedures. For instance, this would enable Isabelle/HOL to automatically derive the optimal induction axiom for procedure *groundterm* (using *member.list*) that currently needs to be supplied by the user.

Chapter 8

Conclusions

Verification of programs is a complex and challenging task. In order to cope with the complexity, theorem provers have been developed with varying degrees of automation. While some theorem provers focus on *proof checking* (i. e., verifying that an alleged proof is indeed a proof), our focus lies on the development of highly automated theorem provers, i. e., on the *computation of proofs*.

First-order theorem provers have achieved a remarkable degree of automation that ranges from automated termination analysis over the automatic choice of promising induction axioms to automated proofs of the base and step cases of an inductive proof. The hypothesis of our work is that such a high degree of automation can be maintained for the verification of *second-order* programs. Second-order programs are a convenient feature: They allow for a more elegant implementation of many algorithms and often such a second-order program can be verified with fewer auxiliary lemmas than the less natural first-order program.

As the practical evaluation of our approach shows, the methods we propose indeed automate significant parts of the verification process. Our contributions are:

1. A generic approach to analyze the dynamic call structure of second-order programs. We introduced so-called *quantification procedures* to reason about the function calls made by second-order procedures. Furthermore, some "canonical" quantification procedures for data structures capture the nesting of data structures.

2. To automate termination proofs of procedures, we extended the approach of *argument-bounded functions* [86, 96]. In this approach, termination of a procedure is shown by proving that the structural size of (some) parameters decreases in recursive calls.

   Our extension additionally considers *components* of types (i. e., substructures of a compound data structure). This extension allows our

approach to prove termination of several purely first-order procedures (e. g., *msort* and *termlist.size*) that cannot be handled by the original approach in [86, 96]. Since our extended method subsumes the original approach, our method solves all the termination problems that the original approach is able to solve.

3. Using our novel notion of *call-boundedness*, termination of procedures with second-order recursion can be proved automatically for the procedures considered in the literature.

4. From a terminating procedure we uniformly synthesize induction axioms that can be used to inductively prove properties of the procedure. These induction axioms are optimized using the results of the termination analysis. Existing optimization techniques from termination analysis via argument-bounded functions [85, 89] are applicable and lead to more powerful induction axioms. For procedures that are defined by second-order recursion we developed similar optimization techniques.

   Such optimizations result in induction axioms that are more general than the non-optimized induction axioms. This has two advantages: Firstly, it simplifies inductive proofs wrt. these induction axioms. Secondly, it facilitates the job of an *induction heuristic* [83, 85], because the optimization usually reveals identical recursion structures of different procedures.

5. Proof goals for the verification of second-order programs typically contain λ-expressions and second-order quantification procedures. We developed *evaluation rules* that simplify such proof goals to maintain the high level of automation of proof techniques that have been developed for the verification of first-order programs.

Our main competitors wrt. *automated* program verification are ACL2 and Isabelle/HOL. ACL2 does not support second-order recursion. The user needs to make a detour via manually instantiated second-order procedures with recursion structures that complicate the verification process, for instance, because of mutual recursion. Proving properties about mutually recursive procedures is a cumbersome task so that the ACL2 developers suggest to avoid this if possible [65]. Isabelle/HOL does not offer a real programming language for the object program to be verified. Consequently, there is no support to reason about required function calls. For each second-order procedure the user must state and prove theorems that describe the behavior of an interpreter when executing these procedures. This may lead to suboptimal induction axioms that are impractical for subsequent proofs [62].

Our approach to verify second-order programs (as described in this thesis) is not the first approach that was tried out in our verification tool

✓eriFun. An earlier approach [19] attempted "to remain as first-order as possible". This turned out to be disadvantageous both from the system developer's and from the user's view:

For the implementation, it was inconvenient to continuously having to translate between the higher-order presentation and the internal first-order representation. The gain for the user was mainly that algorithms could be phrased in a natural higher-order style. However, these algorithms were internally translated into a first-order representation so that second-order recursion became mutual recursion. Practically, the resulting system was only useful when procedures *without* second-order recursion were involved. This unsatisfactory result motivated our development of a theorem prover with dedicated support of second-order recursion so that algorithms can be both stated *and* verified in a natural way.

At first glance, one might wonder why we confine ourselves to verifying *second-order* programs. As discussed in Chapter 2, this restriction is a pragmatic decision: A second-order programming language is already so expressive that a large collection of real-world algorithms can be naturally implemented.[1] At the same time, our restricted second-order language avoids issues regarding the semantics of programs (e. g., undecidability of the equality of functions when interpreting expressions such as $f \in g :: \varepsilon$).

However, our approach easily generalizes to certain *higher-order* programs: For instance, procedure *treemap* (cf. Figure 4.10 on p. 124) can be regarded as a higher-order procedure. Similarly, procedure *map* (which is called by *treemap*) can be a higher-order procedure. Procedure *map* is call-bounded, because it applies a function to the elements of a list. The fact whether these elements are *values* of base type or *functions* is irrelevant to termination, because *map* is defined by recursion on a *list* and *treemap* is defined by recursion on a *tree*. Our approach fails, however, if a procedure is defined by recursion on a *function type* (see [76] for an example of such a procedure). We leave the support of non-structural higher-order recursion as future work.

Further research can also investigate the synthesis of auxiliary lemmas that are needed to prove properties of second-order programs. For instance, we observed that inductive proofs often need lemmas about quantification procedures such as

$$\begin{aligned}&\forall p, q : @A \to \mathit{bool},\ k : \mathit{list}[@A].\\ &\ \mathit{if}\{\mathit{forall.list}(\lambda x : @A.\ \mathit{if}\{p(x), q(x), \mathit{true}\},\ k),\\ &\quad \mathit{if}\{\mathit{forall.list}(p, k),\ \mathit{forall.list}(q, k),\ \mathit{true}\},\\ &\quad \mathit{true}\}\,.\end{aligned}$$

[1] In fact, all the examples of higher-order recursion from the literature on higher-order theorem provers can already be captured by second-order recursion. For instance, to define procedure *groundterm* it suffices to consider *every* as a second-order procedure.

Similar lemmas may be required to reason about an application of *forall.list* to a λ-expression that contains a *case*-expression instead of an *if*-expression. So far we rely on the user to define these lemmas. However, if the user chooses a suboptimal implementation (wrt. verification) and additionally uses procedures *every* and *some*, it would be useful to automate the synthesis of the corresponding auxiliary lemmas.

Finally, it would be interesting to synthesize lemmas about second-order procedures by *generalization*. This is currently supported well for first-order programs (see [10, 11], for instance). Ongoing research at the University of Edinburgh by Moa Johansson suggests that generalization in the higher-order case poses challenges that go way beyond the difficulties encountered in the first-order case. Still, we think that certain progress can be achieved. For example, the proof of lemma "termsize(t) $\leq$ termsize(σ(t))" (cf. Appendix A.2) gets stuck with goal term

$$\begin{aligned}&\neg\, listsum(map(termsize, args(t))) > \\ &\quad listsum(map(\lambda s : term[@\,V, @F].\, termsize(apply.subst(\sigma, s)), \\ &\qquad\qquad args(t)))\end{aligned}$$

and induction hypothesis

$$\begin{aligned}&forall.list(\lambda s : term[@\,V, @F]. \\ &\qquad \neg\, termsize(s) > termsize(apply.subst(\sigma, s)),\ args(t))\,.\end{aligned}$$

Generalizing this goal would lead to exactly the lemma that is required to finish the proof, namely

$$\begin{aligned}&\forall f : @A \to \mathbb{N},\ g : @A \to @A,\ k : list[@A]. \\ &if\{forall.list(\lambda x : @A. \neg\, f(x) > f(g(x)),\ k), \\ &\quad \neg\, listsum(map(f, k)) > listsum(map(\lambda x : @A.\, f(g(x))),\ k), \\ &\quad true\}\,.\end{aligned}$$

Appendix A

Examples

This chapter lists the source code of the following case studies:

- Quicksort (Section A.1)
- Term Operations (Section A.2 on p. 200)
- Lisp Interpreter (Section A.3 on p. 206)
- Variadic Trees (Section A.4 on p. 213)

Implementations of auxiliary procedures that have been defined previously (e. g., procedure + in Figure 1.6 on p. 11) are abbreviated by "...".

A.1 Quicksort

```
procedure [infixr,20] +(x, y : ℕ) : ℕ <= ...

procedure [infixr,30] <>(k, l : list[@A]) : list[@A] <= ...

procedure filter(p : @A → bool,
                 k : list[@A]) : list[@A] <= ...


procedure qsort(k : list[ℕ]) : list[ℕ] <=
if ?empty(k)
  then empty
  else qsort(filter(λ n : ℕ ¬ n > hd(k), tl(k)))
       <> hd(k) :: qsort(filter(λ n : ℕ n > hd(k), tl(k)))
end_if
```

```
procedure ordered(k : list[ℕ]) : bool <=
if ?empty(k)
  then true
  else if ?empty(tl(k))
         then true
         else if hd(k) > hd(tl(k))
                then false
                else ordered(tl(k))
              end_if
       end_if
end_if

procedure [infix*,50] #(i : @ITEM, k : list[@ITEM]) : ℕ <=
if ?empty(k)
  then 0
  else if i = hd(k)
         then ⁺(i # tl(k))
         else i # tl(k)
       end_if
end_if

lemma <> is associative <= ∀ x, y, z : list[@ITEM]
  (x <> y) <> z = x <> y <> z

lemma filter is correct <= ∀ k : list[@ITEM], p : @ITEM → bool
  forall.list(p, filter(p, k))

lemma filter preserves predicates <=
  ∀ k : list[@ITEM], p, q : @ITEM → bool
  if{forall.list(p, k), forall.list(p, filter(q, k)), true}

lemma <> and :: preserve predicates <=
  ∀ k, l : list[@ITEM], p : @ITEM → bool, x : @ITEM
  if{p(x),
    if{forall.list(p, k),
      if{forall.list(p, l), forall.list(p, k <> x :: l), true},
      true},
    true}

lemma qsort preserves predicates <=
  ∀ k : list[ℕ], p : ℕ → bool
  if{forall.list(p, k), forall.list(p, qsort(k)), true}
```

```
lemma qsort sorts lemma <= ∀ k, l : list[ℕ], m : ℕ
  if{ordered(k),
    if{ordered(l),
      if{forall.list(λ n : ℕ ¬ n > m, k),
        if{forall.list(λ n : ℕ n > m, l),
                       ordered(k <> m :: l),
                       true},
        true},
      true},
    true}

lemma qsort sorts <= ∀ k : list[ℕ]
  ordered(qsort(k))

lemma + is commutative <= ∀ x, y : ℕ
  x + y = y + x

lemma occurs append <= ∀ i : @ITEM, k, l : list[@ITEM]
  i # (k <> l) = i # k + i # l

lemma occurs filter <=
  ∀ x : @ITEM, k : list[@ITEM], p : @ITEM → bool
  x # k = x # filter(p, k) + x # filter(λ y : @ITEM ¬ p(y), k)

lemma qsort permutes <= ∀ k : list[ℕ], n : ℕ
  n # k = n # qsort(k)
```

Statistics:

- 6 user-defined procedures, all termination proofs without user interaction
- 11 lemmas (2 main lemmas + 9 auxiliary lemmas)
- no user interaction in proofs

A.2 Terms

```
procedure [infixr,20] +(x, y : ℕ) : ℕ <=

procedure [infixr,30] <>(k, l : list[@A]) : list[@A] <= ...

procedure map(f : @A → @B, k : list[@A]) : list[@B] <= ...

procedure [outfix] |(k : list[@ITEM]) : ℕ <=
if ?empty(k)
  then 0
  else +(| tl(k) |)
end_if

procedure [infix*,10] ∈(i : @ITEM, l : list[@ITEM]) : bool <=
if ?empty(l)
  then false
  else if i = hd(l)
         then true
         else i ∈ tl(l)
       end_if
end_if

procedure sum(k : list[ℕ]) : ℕ <=
if ?empty(k)
  then 0
  else hd(k) + sum(tl(k))
end_if

procedure flatten(k : list[list[@ITEM]]) : list[@ITEM] <=
if ?empty(k)
  then empty
  else hd(k) <> flatten(tl(k))
end_if

procedure lookup(key : @KEY, alist : list[pair[@KEY, @ITEM]],
                 default : @ITEM) : @ITEM <=
if ?empty(alist)
  then default
  else if key = fst(hd(alist))
         then snd(hd(alist))
         else lookup(key, tl(alist), default)
       end_if
end_if
```

```
procedure groundterm(t : term[@V, @F]) : bool <=
case t of
  var : false,
  apply : forall.list(groundterm, args(t))
end_case

procedure subterm(r, t : term[@V, @F]) : bool <=
if r = t
  then true
  else case t of
         var : false,
         apply : ¬ forall.list(
                     λ s : term[@V, @F] ¬ subterm(r, s),
                     args(t))
       end_case
end_if

procedure termsize(t : term[@V, @F]) : ℕ <=
case t of
  var : 1,
  apply : +(sum(map(termsize, args(t))))
end_case

procedure variables(t : term[@V, @F]) : list[@V] <=
case t of
  var : vsym(t) :: empty,
  apply : flatten(map(variables, args(t)))
end_case

procedure wellformed(t : term[@V, @F],
                     arity : list[pair[@F, ℕ]]) : bool <=
case t of
  var : true,
  apply :
    if | args(t) | = lookup(fsym(t), arity, 0)
     then forall.list(λ s : term[@V, @F] wellformed(s, arity),
                      args(t))
     else false
    end_if
end_case
```

```
procedure apply.subst(σ : list[pair[@V, term[@V, @F]]],
                      t : term[@V, @F]) : term[@V,@F] <=
case t of
  var : lookup(vsym(t), σ, t),
  apply :
    apply(fsym(t), map(λ s : term[@V, @F] apply.subst(σ, s),
                       args(t)))
end_case

lemma + is associative <= ∀ x, y, z : ℕ
  (x + y) + z = x + y + z

lemma + is commutative <= ∀ x, y : ℕ
  x + y = y + x

lemma x ≠ 0 ∧ x ≤ y → ⁻(x) ≤ y <=
  ∀ x, y : ℕ
  if{?0(x), true, if{x > y, true, ¬ ⁻(x) > y}}

lemma x ≤ y → x ≤ y + z <= ∀ x, y, z : ℕ
  if{x > y, true, ¬ x > y + z}

lemma a ≤ b ∧ c ≤ d → a + c ≤ b + d <=
  ∀ a, b, c, d : ℕ
  if{a > b, true, if{c > d, true, ¬ a + c > b + d}}

lemma <> is associative <= ∀ x, y, z : list[@ITEM]
  (x <> y) <> z = x <> y <> z

lemma forall.list(∇p, k) → ∇forall.list(p, k) <=
  ∀ p : @A → bool, k : list[@A]
  if{forall.list(∇p, k), ∇forall.list(p, k), true}

lemma forall.list(p → q, k) ∧ forall.list(p, k) →
      forall.list(q, k) <=
  ∀ p, q : @A → bool, k : list[@A]
  if{forall.list(λ x : @A if{p(x), q(x), true}, k),
    if{forall.list(p, k), forall.list(q, k), true},
    true}
```

```
lemma forall.list(p → q, k) ∧ exists.list(p, k) →
      exists.list(q, k) <=
  ∀ p, q : @A → bool, k : list[@A]
  if{forall.list(λ x : @A if{p(x), q(x), true}, k),
    if{forall.list(λ x : @A ¬ p(x), k),
      true,
      ¬ forall.list(λ x : @A ¬ q(x), k)},
    true}

lemma n ∈ k <> l ↔ (n ∈ k ∨ n ∈ l) <=
  ∀ n : @ITEM, k, l : list[@ITEM]
  if{n ∈ k <> l,
    if{n ∈ k, true, n ∈ l},
    if{n ∈ k, false, ¬ n ∈ l}}

lemma forall.list(p, map(f, k)) ↔ forall.list(p ∘ f, k) <=
  ∀ k : list[@A], f : @A → @B, p : @B → bool
  if{forall.list(λ x : @A p(f(x)), k),
    forall.list(p, map(f, k)),
    ¬ forall.list(p, map(f, k))}

lemma | map(f, k) | = | k | <= ∀ f : @A → @B, k : list[@A]
  | map(f, k) | = | k |

lemma map(id, k) = k <= ∀ k : list[@A], f : @A → @A
  if{forall.list(λ x : @A f(x) = x, k), map(f, k) = k, true}

lemma map(g, map(f, k)) = map(g ∘ f, k) <=
  ∀ k : list[@A], f : @A → @B, g : @B → @C
  map(g, map(f, k)) = map(λ x : @A g(f(x)), k)

lemma x ∉ flatten(map(f, k)) <=
  ∀ x : @A, f : @B → list[@A], k : list[@B]
  if{forall.list(λ z : @B ¬ x ∈ f(z), k),
    ¬ x ∈ flatten(map(f, k)),
    true}

lemma flatten(map(f, k)) = empty <=
  ∀ k : list[@A], f : @A → list[@B]
  if{forall.list(λ x : @A ?empty(f(x)), k),
    ?empty(flatten(map(f, k))),
    true}
```

```
lemma exists.list(x ∈ f(_), k) → x ∈ flatten(map(f, k)) <=
  ∀ k : list[@A], f : @A → list[@B], x : @B
  if{forall.list(λ a : @A ¬ x ∈ f(a), k),
    true,
    x ∈ flatten(map(f, k))}

lemma exists.list(f(_) ≥ n, k) → sum(map(f, k)) ≥ n <=
  ∀ k : list[@A], f : @A → ℕ, n : ℕ
  if{forall.list(λ x : @A n > f(x), k),
    true,
    ¬ n > sum(map(f, k))}

lemma sum(map(f, k)) ≤ sum(map(f ∘ g, k)) <=
  ∀ k : list[@A], f : @A → ℕ, g : @A → @A
  if{forall.list(λ x : @A ¬ f(x) > f(g(x)), k),
    ¬ sum(map(f, k)) > sum(map(λ x : @A f(g(x)), k)),
    true}

lemma subterm is reflexive <= ∀ x, y : term[@V, @F]
  if{x = y, subterm(x, y), true}

lemma subterm is transitive <= ∀ x, y, z : term[@V, @F]
  if{subterm(x, y),
    if{subterm(y, z), subterm(x, z), true},
    true}

lemma subterm(r, t) → termsize(r) ≤ termsize(t) <=
  ∀ r, t : term[@V, @F]
  if{subterm(r, t), ¬ termsize(r) > termsize(t), true}

lemma groundterm(t) → ¬ subterm(v, t) <= ∀ v, t : term[@V, @F]
  if{?var(v), if{groundterm(t), ¬ subterm(v, t), true}, true}

lemma groundterm(t) → variables(t)=empty <= ∀ t : term[@V, @F]
  if{groundterm(t), ?empty(variables(t)), true}

lemma subterm(v, t) → v ∈ variables(t) <=
  ∀ v, t : term[@V, @F]
  if{?var(v),
    if{subterm(v, t), vsym(v) ∈ variables(t), true},
    true}
```

```
lemma groundterm(q) → σ(q) = q <=
  ∀ q : term[@V, @F], σ : list[pair[@V, term[@V, @F]]]
  if{groundterm(q), apply.subst(σ, q) = q, true}

lemma v ∉ variables(r) → v ∉ variables(t{v/r}) <=
  ∀ v : @V, r, t : term[@V, @F]
  if{v ∈ variables(r),
    true,
    ¬ v ∈ variables(apply.subst(mkpair(v, r) :: empty, t))}

lemma subterm(v, t) → subterm(r, t{v/r}) <=
  ∀ v : @V, r, t : term[@V, @F]
  if{subterm(var(v), t),
    subterm(r, apply.subst(mkpair(v, r) :: empty, t)),
    true}

lemma termsize(t) ≤ termsize(σ(t)) <=
  ∀ σ : list[pair[@V, term[@V, @F]]], t : term[@V, @F]
  ¬ termsize(t) > termsize(apply.subst(σ, t))

lemma wellformed(t{v/r}) <=
  ∀ t, r : term[@V, @F], v : @V, arity : list[pair[@F, ℕ]]
  if{wellformed(t, arity),
    if{wellformed(r, arity),
      wellformed(apply.subst(mkpair(v, r) :: empty, t),
                 arity),
      true},
    true}
```

Statistics:

- 14 user-defined procedures (6 procedures with second-order recursion), all termination proofs without user interaction
- 30 lemmas (11 main lemmas + 19 auxiliary lemmas)
- 2 user interactions in proofs (see Section 7.2)

A.3 Lisp Interpreter

```
structure predefinedSymbol <=
  T, ADD1, SUB1, NUMBERP, LISTP, CONS, CAR, CDR, LIST, EQUAL,
  QUOTE, IF

structure sexpr <=
  nil,
  lispsymbol(lname : predefinedSymbol),
  usersymbol(uname : ℕ),
  number(value : ℕ),
  cons(car : sexpr, cdr : sexpr)

structure result[@TYPE] <=
  ⊥,
  def(val : @TYPE)

procedure [infixr,20] +(x, y : ℕ) : ℕ <= ...

procedure pairlist(args : sexpr, vals : sexpr) : sexpr <=
if ?cons(args)
  then if ?cons(vals)
         then cons(cons(car(args), car(vals)),
                   pairlist(cdr(args), cdr(vals)))
         else cons(cons(car(args), nil),
                   pairlist(cdr(args), nil))
       end_if
  else nil
end_if

procedure get(key : sexpr, alist : sexpr) : sexpr <=
if ?cons(alist)
  then if ?cons(car(alist))
         then if car(car(alist)) = key
                then cdr(car(alist))
                else get(key, cdr(alist))
              end_if
         else nil
       end_if
  else nil
end_if
```

```
procedure mapsx(f : sexpr → result[sexpr],
                x : sexpr) : result[sexpr] <=
if ?cons(x)
  then let f-car := f(car(x)) in
         if ?def(f-car)
           then let f-cdr := mapsx(f, cdr(x)) in
                  if ?def(f-cdr)
                    then def(cons(val(f-car), val(f-cdr)))
                    else ⊥
                  end_if
                end_let
           else ⊥
         end_if
       end_let
  else def(nil)
end_if

// Automatically synthesized quantification procedure.
procedure forall.mapsx(f : sexpr → result[sexpr],
                       p_f : sexpr → bool,
                       x : sexpr) : bool <=
if ?cons(x)
  then if p_f(car(x))
         then if ?def(f(car(x)))
                then forall.mapsx(f, p_f, cdr(x))
                else true
              end_if
         else false
       end_if
  else true
end_if

// Automatically synthesized optimized quantification procedure.
procedure forall-opt.mapsx(p_f : sexpr → bool,
                           x : sexpr) : bool <=
if ?cons(x)
  then if p_f(car(x))
         then forall-opt.mapsx(p_f, cdr(x))
         else false
       end_if
  else true
end_if
```

```
procedure apply(fn : predefinedSymbol,
                args : sexpr) : sexpr <=
case fn of
  ADD1 :
    if ?cons(args)
      then if ?number(car(args))
             then number(+(value(car(args))))
             else nil
           end_if
      else nil
    end_if,
  SUB1 :
    if ?cons(args)
      then if ?number(car(args))
             then if ?+(value(car(args)))
                    then number(-(value(car(args))))
                    else nil
                  end_if
             else nil
           end_if
      else nil
    end_if,
  NUMBERP :
    if ?cons(args)
      then if ?number(car(args))
             then lispsymbol(T)
             else nil
           end_if
      else nil
    end_if,
  LISTP :
    if ?cons(args)
      then if ?cons(car(args))
             then lispsymbol(T)
             else nil
           end_if
      else nil
    end_if,
```

```
  CONS :
    if ?cons(args)
      then if ?cons(cdr(args))
             then cons(car(args), car(cdr(args)))
             else nil
           end_if
      else nil
    end_if,
  CAR :
    if ?cons(args)
      then if ?cons(car(args))
             then car(car(args))
             else nil
           end_if
      else nil
    end_if,
  CDR :
    if ?cons(args)
      then if ?cons(car(args))
             then cdr(car(args))
             else nil
           end_if
      else nil
    end_if,
  LIST :
    if ?cons(args)
      then args
      else nil
    end_if,
  EQUAL :
    if ?cons(args)
      then if ?cons(cdr(args))
             then if car(args) = car(cdr(args))
                    then lispsymbol(T)
                    else nil
                  end_if
             else nil
           end_if
      else nil
    end_if,
  other : nil
end_case
```

```
procedure eval(expr, va, fa : sexpr, n : ℕ) : result[sexpr] <=
if ?0(n)
  then ⊥
  else
    case expr of
      nil : def(nil),
      lispsymbol :
        case lname(expr) of
          T : def(lispsymbol(T)),
          other : def(nil)
        end_case,
      usersymbol : def(get(expr, va)),
      number : def(expr),
      cons :
        case car(expr) of
          nil : def(nil),
          lispsymbol :
            case lname(car(expr)) of
              QUOTE :
                if ?cons(cdr(expr))
                  then def(car(cdr(expr)))
                  else def(nil)
                end_if,
              IF :
                if ?cons(cdr(expr))
                  then
                    let switch := eval(car(cdr(expr)),
                                       va, fa, n) in
                      if ?def(switch)
                        then
                          if ?cons(cdr(cdr(expr)))
                            then
                              if ?nil(val(switch))
                                then
                                  if ?cons(cdr(cdr(cdr(expr))))
                                    then eval(car(cdr(cdr(
                                                cdr(expr)))),
                                              va, fa, n)
                                    else def(nil)
                                  end_if
                                else eval(car(cdr(cdr(expr))),
                                          va, fa, n)
                              end_if
                            else def(nil)
```

```
                        end_if
                      else ⊥
                    end_if
                  end_let
                else def(nil)
              end_if,
            other :
              let args := mapsx(λ arg : sexpr
                                eval(arg, va, fa, n),
                                cdr(expr)) in
                if ?def(args)
                  then def(apply(lname(car(expr)),
                                 val(args)))
                  else ⊥
                end_if
              end_let
          end_case,
        usersymbol :
          let f := get(car(expr), fa) in
            if ?cons(f)
              then let args := mapsx(λ arg : sexpr
                                     eval(arg, va, fa, n),
                                     cdr(expr)) in
                     if ?def(args)
                       then eval(cdr(f),
                                 pairlist(car(f), val(args)),
                                 fa,
                                 ⁻(n))
                       else ⊥
                     end_if
                   end_let
              else def(nil)
            end_if
          end_let,
        other : def(nil)
      end_case
  end_case
end_if

lemma + is associative <= ∀ x, y, z : ℕ
  (x + y) + z = x + y + z

lemma + is commutative <= ∀ x, y : ℕ
  x + y = y + x
```

```
lemma mapsx(f, x) = mapsx(g, x) <=
  ∀ f, g : sexpr → result[sexpr], x : sexpr
  if{?def(mapsx(f, x)),
    if{forall-opt.mapsx(λ z : sexpr
                        if{?⊥(f(z)), true, f(z) = g(z)},
                        x),
      mapsx(f, x) = mapsx(g, x),
      true},
    true}

lemma increase resources <= ∀ n, k : ℕ, fa, va, expr : sexpr
  if{?def(eval(expr, va, fa, n)),
    eval(expr, va, fa, n) = eval(expr, va, fa, n + k),
    true}
```

Statistics:

- 6 user-defined procedures (1 procedure with second-order recursion), all termination proofs without user interaction
- 4 lemmas (1 main lemma + 3 auxiliary lemmas)
- 1 user interaction in proof of lemma `increase resources` (*Apply Equation* `mapsx(f, x) = mapsx(g, x)`)

A.4 Variadic Trees

```
structure tree[@A] <=
  leaf(val : @A),
  branch(children : list[tree[@A]])

procedure [infixr,10] <>(k, l : list[@A]) : list[@A] <= ...

procedure map(f : @A → @B, k : list[@A]) : list[@B] <= ...

procedure some(p : @A → bool, k : list[@A]) : bool <= ...

procedure in.list(x : @ITEM, k : list[@ITEM]) : bool <=
if ?empty(k)
  then false
  else if hd(k) = x
         then true
         else in.list(x, tl(k))
       end_if
end_if

procedure flatten.list(k : list[list[@ITEM]]) : list[@ITEM] <=
if ?empty(k)
  then empty
  else hd(k) <> flatten.list(tl(k))
end_if

procedure flatten.tree(t : tree[@A]) : list[@A] <=
case t of
  leaf : val(t) :: empty,
  branch : flatten.list(map(flatten.tree, children(t)))
end_case

procedure in.tree(x : @ITEM, t : tree[@ITEM]) : bool <=
case t of
  leaf : val(t) = x,
  branch : some(λ s : tree[@ITEM] in.tree(x, s), children(t))
end_case

lemma <> is associative <= ∀ x, y, z : list[@A]
  (x <> y) <> z = x <> y <> z
```

```
lemma forall.list(p → q, k) ∧ some(p, k) → some(q, k) <=
  ∀ p, q : @A → bool, k : list[@A]
  if{forall.list(λ x : @A if{p(x), q(x), true}, k),
    if{some(p, k), some(q, k), true},
    true}

lemma in.list(x, k) → in.list(x, k <> l) <=
  ∀ x : @ITEM, k, l : list[@ITEM]
  if{in.list(x, k), in.list(x, k <> l), true}

lemma in.list(x, l) → in.list(x, k <> l) <=
  ∀ x : @ITEM, k, l : list[@ITEM]
  if{in.list(x, l), in.list(x, k <> l), true}

lemma in.list(x, k <> l) → in.list(x, k) ∨ in.list(x, l) <=
  ∀ x : @ITEM, k, l : list[@ITEM]
  if{in.list(x, k <> l),
    if{in.list(x, k), true, in.list(x, l)},
    true}

lemma in flattened map list <=
  ∀ k : list[@A], f : @A → list[@B], x : @B
  if{some(λ a : @A in.list(x, f(a)), k),
    in.list(x, flatten.list(map(f, k))),
    ¬ in.list(x, flatten.list(map(f, k)))}

lemma in.tree(x, t) → in.list(x, flatten(t)) <=
  ∀ x : @ITEM, t : tree[@ITEM]
  if{in.tree(x, t), in.list(x, flatten.tree(t)), true}

lemma in.list(x, flatten(t)) → in.tree(x, t) <=
  ∀ x : @ITEM, t : tree[@ITEM]
  if{in.list(x, flatten.tree(t)), in.tree(x, t), true}
```

Statistics:

- 7 user-defined procedures (2 procedures with second-order recursion), all termination proofs without user interaction
- 8 lemmas (2 main lemmas + 6 auxiliary lemmas)
- 1 user interaction in the proof of lemma "`in.list(x, flatten(t)) → in.tree(x, t)`": *Use Lemma* with "`forall.list(p → q, k) ∧ some(p, k) → some(q, k)`"

Bibliography

[1] `http://www.verifun.org/`.

[2] `http://www.cs.utexas.edu/users/moore/acl2/`.

[3] `http://coq.inria.fr/`.

[4] `http://hol.sourceforge.net/`.

[5] `http://isabelle.in.tum.de/`.

[6] `http://www.nuprl.org/`.

[7] `http://pvs.csl.sri.com/`.

[8] Andreas Abel. Termination and productivity checking with continuous types. In M. Hofmann, editor, *Proceedings of International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 2701 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2003.

[9] Andreas Abel. Termination checking with types. *RAIRO – Theoretical Informatics and Applications*, 38(4):277–319, 2004. Special Issue: Fixed Points in Computer Science (FICS'03).

[10] Markus Aderhold. Formula generalization in ✓`eriFun`. Diploma thesis, Technische Universität Darmstadt, Germany, 2004.

[11] Markus Aderhold. Improvements in formula generalization. In Frank Pfenning, editor, *Proceedings of the 21st International Conference on Automated Deduction (CADE)*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 231–246. Springer-Verlag, 2007.

[12] Markus Aderhold. Automated termination analysis for programs with second-order recursion. Technical report, Technische Universität Darmstadt, 2009.

[13] Markus Aderhold, Christoph Walther, Daniel Szallies, and Andreas Schlosser. A fast disprover for ✓`eriFun`. In Wolfgang Ahrendt, Peter Baumgartner, and Hans de Nivelle, editors, *Proceedings of the 3rd*

Workshop on Disproving, IJCAR 2006, pages 59–69, Seattle (WA), USA, 2006.

[14] Peter B. Andrews. Classical type theory. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 15, pages 965–1007. Elsevier Science, 2001.

[15] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Kluwer Academic Publishers, 2002.

[16] Takahito Aoto and Toshiyuki Yamada. Termination of simply typed term rewriting by translation and labelling. In R. Nieuwenhuis, editor, *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA)*, volume 2706 of *Lecture Notes in Computer Science*, pages 380–394. Springer-Verlag, 2003.

[17] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

[18] Raymond Aubin. Mechanizing structural induction, part II: Strategies. *Theoretical Computer Science*, 9(3):347–362, October 1979.

[19] Jens Auer. Erweiterung des ✓`eriFun`-Systems um Funktionen höherer Ordnung und gegenseitige Rekursion. Diploma thesis, Technische Universität Darmstadt, Germany, 2005.

[20] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*, chapter 3. Cambridge University Press, 1998.

[21] Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu. Defining and reasoning about recursive functions: A practical tool for the Coq proof assistant. In *Proceedings of FLOPS-06*, volume 3945 of *Lecture Notes in Computer Science*, pages 114–129, 2006.

[22] Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. Practical inference for typed-based termination in a polymorphic setting. In *Proceedings of 7th International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 3461 of *Lecture Notes in Computer Science*, pages 71–85. Springer-Verlag, 2005.

[23] Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. CIC$\hat{}$: Type-based termination of recursive definitions in the calculus of inductive constructions. In *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 4246 of *Lecture Notes in Computer Science*, pages 257–271. Springer-Verlag, 2006.

[24] Gilles Barthe, Benjamin Grégoire, and Colin Riba. Type-based termination with sized products. In M. Kaminski and S. Martini, editors, *Proceedings of the 22nd international workshop on Computer Science Logic*, volume 5213 of *Lecture Notes in Computer Science*, pages 493–507. Springer-Verlag, 2008.

[25] Bernhard Beckert, Martin Giese, Reiner Hähnle, Vladimir Klebanov, Philipp Rümmer, Steffen Schlager, and Peter H. Schmitt. The KeY System 1.0 (deduction component). In Frank Pfenning, editor, *Proceedings of the 21st International Conference on Automated Deduction (CADE)*, volume 4603 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[26] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[27] Ulrich Berger, Stefan Berghofer, Pierre Letouzey, and Helmut Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82(1):25–49, 2006.

[28] Stefan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In *Software Engineering and Formal Methods*, pages 230–239. IEEE Computer Society, 2004.

[29] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[30] Frédéric Blanqui and Colin Riba. Combining typing and size constraints for checking the termination of higher-order conditional rewrite systems. In Miki Hermann and Andrei Voronkov, editors, *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 4246 of *Lecture Notes in Computer Science*, pages 105–119. Springer-Verlag, 2006.

[31] Richard J. Boulton and Konrad Slind. Automatic derivation and application of induction schemes for mutually recursive functions. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. Moniz Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Proceedings of the First International Conference on Computational Logic*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 629–643. Springer-Verlag, 2000.

[32] Robert S. Boyer, David M. Goldschlag, Matt Kaufmann, and J Strother Moore. Functional instantiation in first-order logic. In

Vladimir Lifschitz, editor, *Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.

[33] Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, Inc., 1979.

[34] Robert S. Boyer and J. Strother Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of the ACM*, 31(3):441–458, 1984.

[35] Bruno Buchberger and Adrian Crăciun. Algorithm synthesis by lazy thinking: Using problem schemes. In D. Petcu, V. Negru, D. Zaharie, and T. Jebelean, editors, *Proceedings of SYNASC 2004, 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 90–106, 2004.

[36] Alan Bundy. The automation of proof by mathematical induction. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 13, pages 845–911. Elsevier Science, 2001.

[37] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling: Meta-Level Guidance for Mathematical Reasoning*, chapter 3. Number 56 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2005.

[38] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[39] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2-3):261–300, 2001.

[40] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1985.

[41] Lucas Dixon. *A Proof Planning Framework For Isabelle*. PhD thesis, University of Edinburgh, 2005.

[42] John D. Erickson. *Generalization, Lemma Generation, and Induction in ACL2*. PhD thesis, University of Texas at Austin, USA, May 2008.

[43] Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proceedings of IJCAR-2006*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 281–286. Springer-Verlag, 2006.

[44] Jürgen Giesl, Stephan Swiderski, Peter Schneider-Kamp, and René Thiemann. Automated termination analysis for Haskell: From term rewriting to programming languages. In *Proceedings of the 17th International Conference on Rewriting Techniques and Applications (RTA)*, volume 4098 of *Lecture Notes in Computer Science*, pages 297–312. Springer-Verlag, 2006.

[45] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 3452 of *Lecture Notes in Artificial Intelligence*, pages 301–331. Springer-Verlag, 2005.

[46] Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp. Proving termination by bounded increase. In Frank Pfenning, editor, *Proceedings of the 21st International Conference on Automated Deduction (CADE)*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 443–459. Springer-Verlag, 2007.

[47] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[48] Andrew D. Gordon. A tutorial on co-induction and functional programming. In *In Glasgow Functional Programming Workshop*, pages 78–95. Springer-Verlag, 1994.

[49] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993.

[50] David Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1981.

[51] Florian Haftmann and Tobias Nipkow. A code generator framework for Isabelle/HOL. Technical Report 364/07, Department of Computer Science, University of Kaiserslautern, 2007.

[52] Martin Hofmann and Ulrich Schöpp. Pure pointer programs with iteration. In *Proceedings of the 22nd international workshop on Computer Science Logic*, volume 5213 of *Lecture Notes In Computer Science*, pages 79–93. Springer-Verlag, 2008.

[53] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of Symposium on Principles of Programming Languages*, pages 410–423. ACM, 1996.

[54] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[55] Jean-Pierre Jouannaud and Albert Rubio. Polymorphic higher-order recursive path orderings. *Journal of the ACM*, 54(1), 2007.

[56] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.

[57] Matt Kaufmann and J Strother Moore. Design goals for ACL2. Technical Report 101, University of Texas at Austin, USA, August 1994.

[58] Matt Kaufmann and J Strother Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, 1997.

[59] Matt Kaufmann and J Strother Moore. An ACL2 tutorial. In O. Ait Mohamed, C. Muñoz, and S. Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLS)*, volume 5170 of *Lecture Notes in Computer Science*, pages 17–21. Springer-Verlag, 2008.

[60] Michael Kohlhase. Beweissysteme mit Logiken höherer Stufe. In Karl-Hans Bläsius and Hans-Jürgen Bürckert, editors, *Deduktionssysteme – Automatisierung des logischen Denkens*, chapter VI. Oldenbourg, 2nd edition, 1992.

[61] Alexander Krauss. Defining recursive functions in Isabelle/HOL 2007. Proceedings of the Isabelle Workshop 2007, CADE-21, Bremen, online available at `http://homepages.inf.ed.ac.uk/ldixon/events/isabelle-ws-07/isabelle-07.pdf`, July 2007.

[62] Alexander Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. PhD thesis, Institut für Informatik, Technische Universität München, Germany, 2009. To appear.

[63] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. *ACM Symposium on Principles of Programming Languages*, 36(3):81–92, 2001.

[64] N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1–2):159–172, 1991.

[65] J Strother Moore. Finite set theory in ACL2. In *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLS)*, volume 2152 of *Lecture Notes in Computer Science*, pages 313–328. Springer-Verlag, 2001.

[66] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[67] Sam Owre, John Rushby, and Natarajan Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[68] Sam Owre and Natarajan Shankar. The PVS prelude library. Technical Report SRI-CSL-03-01, SRI International, March 2003.

[69] Sam Owre, Natarajan Shankar, John M. Rushby, and David W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, November 2001.

[70] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[71] Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. The MIT Press, second edition edition, 1987.

[72] Andreas Schlosser, Christoph Walther, Michael Gonder, and Markus Aderhold. Context dependent procedures and computed types in ✓`eriFun`. In *Proceedings of 1st Workshop Programming Languages meet Program Verification*, volume 174 of *ENTCS*, pages 61–78, 2007.

[73] Stephan Schweitzer. *Symbolische Auswertung und Heuristiken zur Verifikation funktionaler Programme*. Doctoral dissertation, Technische Universität Darmstadt, Germany, 2007.

[74] Damien Sereni. Size-change termination of higher-order functional programs. Research Report RR-04-20, Programming Research Group, Oxford University Computing Laboratory, October 2004.

[75] Damien Sereni. Termination analysis and call graph construction for higher-order functional programs. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming (ICFP 2007)*, pages 71–84, New York, NY, USA, 2007. ACM.

[76] Damien Sereni and Neil D. Jones. Termination analysis of higher-order functional programs. In Kwangkeun Yi, editor, *Proceedings of the Third Asian Symposium on Programming Languages and Systems (APLAS 2005)*, volume 3780 of *Lecture Notes in Computer Science*, pages 281–297. Springer-Verlag, 2005.

[77] Natarajan Shankar, Sam Owre, John M. Rushby, and David W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, November 2001.

[78] Moritz Sinn. Implementierung, Integration und Evaluation eines ACI-Matchingverfahrens in √`eriFun`. Bachelor thesis, Technische Universität Darmstadt, Germany, 2007.

[79] Konrad Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, Technische Universität München, 1999.

[80] Daniel Szallies. Ein Werkzeug zur automatischen Widerlegung von Aussagen in √`eriFun`. Diploma thesis, Technische Universität Darmstadt, 2006.

[81] Harvey Tuch and Gerwin Klein. A unified memory model for pointers. In Geoff Sutcliffe and Andrei Voronkov, editors, *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 3835 of *Lecture Notes in Computer Science*, pages 474–488, December 2005.

[82] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[83] Christoph Walther. Computing induction axioms. In Andrei Voronkov, editor, *Proceedings of the 3rd International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 381–392. Springer-Verlag, 1992.

[84] Christoph Walther. Combining induction axioms by machine. In Ruzena Bajcsy, editor, *Proceedings of IJCAI-13*, pages 95–101. Morgan Kaufmann, 1993.

[85] Christoph Walther. Mathematical induction. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 127–228. Oxford University Press, 1994.

[86] Christoph Walther. On proving the termination of algorithms by machine. *Artificial Intelligence*, 71(1):101–157, 1994.

[87] Christoph Walther. Criteria for termination. In S. Hölldobler, editor, *Intellectics and Computational Logic*, pages 361–386. Kluwer Academic Publishers, Dordrecht, 2000.

[88] Christoph Walther. *Semantik und Programmverifikation.* Teubner-Wiley, Leipzig, 2001.

[89] Christoph Walther. Recursion, induction, verification. Lecture notes, Technische Universität Darmstadt, Germany, 2004.

[90] Christoph Walther, Markus Aderhold, and Andreas Schlosser. The $\mathcal{L}$ 1.0 Primer. Technical Report VFR 06/01, Technische Universität Darmstadt, Germany, 2006.

[91] Christoph Walther and Stephan Schweitzer. ✓`eriFun` user guide. Technical Report VFR 02/01, Technische Universität Darmstadt, Germany, 2002.

[92] Christoph Walther and Stephan Schweitzer. About ✓`eriFun`. In Franz Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction (CADE)*, volume 2741 of *Lecture Notes in Computer Science*, pages 322–327. Springer-Verlag, 2003.

[93] Christoph Walther and Stephan Schweitzer. A machine-verified code generator. In M. Y. Vardi and A. Voronkov, editors, *Proceedings of the 10th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 2850 of *Lecture Notes in Artificial Intelligence*, pages 91–106. Springer-Verlag, 2003.

[94] Christoph Walther and Stephan Schweitzer. Reasoning about incompletely defined programs. Technical Report VFR 04/02, Technische Universität Darmstadt, 2004.

[95] Christoph Walther and Stephan Schweitzer. Verification in the classroom. *Journal of Automated Reasoning*, 32(1):35–73, 2004.

[96] Christoph Walther and Stephan Schweitzer. Automated termination analysis for incompletely defined programs. In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 3452 of *Lecture Notes in Artificial Intelligence*, pages 332–346. Springer-Verlag, 2005.

[97] Christoph Walther and Stephan Schweitzer. Reasoning about incompletely defined programs. In G. Sutcliffe and A. Voronkov, editors, *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 3835 of *Lecture Notes in Artificial Intelligence*, pages 427–442. Springer-Verlag, 2005.

[98] Christoph Walther and Stephan Schweitzer. A pragmatic approach to equality reasoning. Technical Report VFR 06/02, Technische Universität Darmstadt, Germany, 2006.

[99] Nathan Wasser. Induction proofs for second-order procedures in ✓`eriFun`. Diploma thesis, Technische Universität Darmstadt, Germany, 2009.

[100] Hongwei Xi. Dependent types for program termination verification. In *Proceedings of 16th IEEE Symposium on Logic in Computer Science*, pages 231–242, 2001.

List of Functions and Types

Index